Computational Science and Engineering in Python

Hans Fangohr

September 21, 2016

Engineering and the Environment
University of Southampton
United Kingdom
fangohr@soton.ac.uk

Outline

- Python prompt
- Functions
- About Python
- Coding style
- Conditionals, if-else
- Sequences
- Loops
- Some things revisited
- Reading and Writing files
- Exceptions
- Printing
- Higher Order Functions

Modules

Default arguments

Namespaces

Python IDEs

List comprehension

Dictionaries

Recursion

Common Computational Tasks

Root finding

Derivatives

Numpy

Higher Order Functions 2: Functional tools

Object Orientation and all that

Numerical Integration

Numpy usage examples

Scientific Python

ODEs

Sympy

Testing

Object Oriented Programming

Some programming languages

What language to learn next?

Python prompt

The Python prompt

- Spyder (or IDLE, or `python` or `python.exe` from shell/Terminal/MS-Dos prompt, or `IPython`)
- Python prompt waits for input:

  ```
  >>>
  ```

- Interactive Python prompt waits for input:

  ```
  In [1]:
  ```

- Read, Evaluate, Print, Loop $\rightarrow$ REPL

Hello World program

Standard greeting:

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

Entered interactively in Python prompt:

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

Or in IPython prompt:

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

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
10
>>> b
20
>>> a + b
30
>>> ab2 = (a + b) / 2
>>> ab2
15
```

Important data types / type()

```
>>> a = 1
>>> type(a)
<class int>              # integer

>>> b = 1.0
>>> type(b)
<class float>            # float

>>> c = '1.0'
>>> type(c)
<class str>              # string

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

Summary useful commands (introspection)

- `print(x)` to display the object `x`
- `type(x)` to determine the type of object `x`
- `help(x)` to obtain the documentation string
- `dir(x)` to display the methods and members of object `x`, or the current name space (`dir()`).

Example:

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

abs(...)
    abs(number) -> number

    Return the absolute value of the argument.
```

Interactive documentation, introspection

```
>>> 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
```

Functions

First use of functions

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

```
def mysum(a,b):
    """Return the sum of parameters a and b.
    Hans Fangohr, fangohr@soton.ac.uk,
    last modified 24/09/2013
    """
    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.
    Hans Fangohr, fangohr@soton.ac.uk,
    last modified 24/09/2013
```

Function terminology

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

- `x` is the *argument* given to the function
- `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.)

Function example

```
def plus42(n):
    """Add 42 to n and return""" # docstring
    l = n + 42                   # body of
                                 # function
    return l

a = 8
b = plus42(a)    # not part of function definition
```

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!)
- docstring provides the specification (contract) of the function's input, output and behaviour
- 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

Given the following two function definitions:

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

def return42():
    return 42
```

we use the Python prompt to explore the difference:

```
>>> 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
```

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
```

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 documentation

There is lots of documentation that you should learn to use:

- Teaching materials on website, including these slides and a text-book like document
  - Online documentation, for example
    - Python home page (http://www.python.org)
    - Pylab/Matplotlib (plotting as in Matlab)
    - Numpy (fast vectors and matrices, (NUMerical PYthon)
    - SciPy (scientific algorithms, `odeint`)
    - Visual Python (3d visualisation)
    - SymPy (Symbolic calculation)
- interactive documentation

Which Python version

- There are currently two versions of Python:
  - Python 2.7 and
  - Python 3.x
- We will use version 3.5 or later
- Python 2.x and 3.x are incompatible although the changes only affect very few commands.
- Write new programs in Python 3 where possible.
- You may have to read / work with Python 2 code at some point.
- See webpages for notes on installation of Python on computers.

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)
```

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` → `1.0`).

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

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

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

- Or make use of Python's future division:

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

If you really want integer division, use `//` instead of `/`:

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

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>
```

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
```

Coding style

Coding 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

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

- This document gives coding conventions for the Python code comprising the standard library in the main Python distribution.
- This style guide evolves over time as additional conventions are identified and past conventions are rendered obsolete by changes in the language itself.
- Many projects have their own coding style guidelines. In the event of any conflicts, such project-specific guides take precedence for that project.
- 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. *"Readability counts"*.
- Sometimes we should *not* follow the style guide, for example:
  - When applying the guideline would make the code less readable, even for someone who is used to reading code that follows this PEP.
  - To be consistent with surrounding code that also breaks it (maybe for historic reasons) – although this is also an opportunity to clean up someone else's mess (in true XP style).

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: `x = 3*a + 4*b` is okay, but also okay to write `x = 3 * a + 4 * b`.
- 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

- Two empty lines between functions
- One import statement per line
- import first standand 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

- Try to follow PEP8 guide, in particular for new code
- Use tools to help us, for example Spyder editor can show PEP8 violations.
  Similar tools/plugins are available for Emacs, Sublime Text, and other editors.
- `pep8` program available to check source code from command line.

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>
```

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
```

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
```

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

```
>>> 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
```

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.

`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
```

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

LAB2

Sequences

Sequences overview

Different types of sequences

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

They share common commands.

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) whereas lists are *mutable* (changeable)
- tuples are usually written using parentheses ($\leftrightarrow$ "round brackets"):

```
>>> 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. allow to 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
     ```

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[4] = '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 i-th element 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.

Conversions

- We can convert any sequence into a tuple using the `tuple` function:

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

- Similarly, the `list` function, converts sequences into lists:

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

- *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>
```

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("This is the {}".format(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("the square of {} is {}"
          .format(i, 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 the `range` iterator

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

```
for i in range(6):
    print("the square of {} is {}"
          .format(i, 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 iterator

- `range` is used to iterate over integer sequences
- `range` has its own type:

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

- We can use iterators in for loops:

```
>>> for i in range(4):
...     print("i={}".format(i))
i=0
i=1
i=2
i=3
```

- We can convert an iterator into a list:

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

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

```
>>> 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` iterator

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

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

Examples:

```
>>> 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
```

(Note that `range` objects are sequences.)

Iterating over sequences with `for`-loop

`for` loop iterates over sequences

Examples:

```
for i in [0, 3, 4, 19]:
    print(i)

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 sequences
```

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):
    result = []
    for k in range(a, b):
        if k == 13:
            pass              # do nothing
        else:
            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]
```

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 iterator
- New: a `while` loop iterates while a condition is fulfilled

Example:

```
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("epsilon is {}".format(eps))
```

Output:

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

Some things revisited

What are variables?

In Python, variables are references (or names) to 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]
```

`id`, `==` and `is`

- 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)` or `a is b` wether `a` and `b` are the *same object*.
- Two different objects can have the *same value*. We check with `==`

See “Equality and identity“, section 3.5

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

```
>>> 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*. Example:

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

xs = [10, 20, 30]
print("xs = {};   ".format(xs), end='')
print("sum(xs)={};   ".format(sum(xs)), end='')
print("xs = {}".format(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 in-built command `sum(xs)`
- 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
```

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.
- See slide 19.

Reading and Writing files

File input/output

It is a (surprisingly) 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

```
>>> f = open('test.txt', 'w')  # Write
>>> f.write("first line\nsecond line")
22        # returns number of chars written
>>> f.close()
```

creates a file `test.txt` that reads

```
first line
second line
```

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

  ```
  f = open('test.txt', 'w')
  ```

  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:

  ```
  f = open('test.txt', 'tw')
  ```

- 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 in input argument.
- Must close file at the end of writing process using `f.close()`.

Reading a text file

We create a file object `f` using

```
>>> f = open('test.txt', 'r')      # Read
```

and have different ways of reading the data:

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

   ```
   >>> f = open('test.txt', 'r')
   >>> lines = f.readlines()
   >>> f.close()
   >>> lines
   ['first line\n', 'second line']
   ```

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

```
>>> f = open('test.txt', 'r')
>>> data = f.read()
>>> f.close()
>>> data
'first line\nsecond line'
```

3. Use text file `f` as an iterable object: process one line in each iteration (important for large files):

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

Opening and *automatic* file closing through context manager

Python provides *context managers* that we use using `with`. For the file access:

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

If we use the file context manager, it will close the file automatically (when the control flows leaves the indented block).

Once you are familiar with file access, we recommend you use this method.

Use case: Reading a file, iterating over lines

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

```
f = open('myfile.txt', 'r')
lines = f.readlines()
f.close()
# some processing of the lines object
```

  `lines` is a list of strings, each representing one line of the file.
- It is good practice to close a file as soon as possible.

- Equivalent example using the context manager:

```
with open('myfile.txt', 'r') as f:
    lines = f.readlines()
# some processing of the lines object
```

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: Take a string and display each word on a separate line:

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

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**

```
fin = open('shopping.txt', 'tr')        # INput File
lines = fin.readlines()
fin.close()      # close file as soon as possible

fout = open('shopping_cost.txt', 'tw') # 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("{:20} {}\n".format(itemname,
                                   totalcost))
fout.close()
```

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

```
1 2 4 67 -34 340
0 45 3 2
17
```

LAB4

Binary files

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

```
f = 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 need it, do learn about the `struct` module.

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:

```
>>> 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 how
  - what exceptions are coming with Python
  - we can “catch” exceptions
  - we can raise (“throw”) exceptions in our code

In-built Python exceptions

Python's in-built exceptions (from documentation)

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
```

```
+-- LookupError
|    +-- IndexError
|    +-- KeyError
+-- MemoryError
+-- NameError
|    +-- UnboundLocalError
+-- OSError
|    +-- BlockingIOError
|    +-- ChildProcessError
|    +-- ConnectionError
|    |    +-- BrokenPipeError
|    |    +-- ConnectionAbortedError
|    |    +-- ConnectionRefusedError
|    |    +-- ConnectionResetError
|    +-- FileExistsError
|    +-- FileNotFoundError
|    +-- InterruptedError
|    +-- IsADirectoryError
|    +-- NotADirectoryError
|    +-- PermissionError
|    +-- ProcessLookupError
```

```
|    +-- TimeoutError
+-- ReferenceError
+-- RuntimeError
|    +-- NotImplementedError
|    +-- RecursionError
+-- SyntaxError
|    +-- IndentationError
|         +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|    +-- UnicodeError
|         +-- UnicodeDecodeError
|         +-- UnicodeEncodeError
|         +-- UnicodeTranslateError
+-- Warning
     +-- DeprecationWarning
     +-- PendingDeprecationWarning
     +-- RuntimeWarning
     +-- SyntaxWarning
     +-- UserWarning
```

```
      +-- FutureWarning
      +-- ImportWarning
      +-- UnicodeWarning
      +-- BytesWarning
      +-- ResourceWarning
```

Somewhat advanced use of Python: We can provide our own exception classes (by inheriting from `Exception`).

Exceptions example

- suppose we try to read data from a file:

```
f = open('myfilename.dat', 'r')
for line in f.readlines():
    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:

```
1  import sys
2  try:
3      f = open('myfilename.txt', 'r')
4  except FileNotFoundError:
5      print("The file couldn't be found. " +
6            "This program stops here.")
7      sys.exit(1)    # a way to exit the program
8
9  for line in f:
10     print(line, end='')
11 f.close()
```

which produces this message:

```
The file couldn't be found. This program stops
↪  here.
```

- The `try` branch (line 3) will be executed.
- Should an `FileNotFoundError` exception be raised, then the except branch (starting line 5) 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 9.
- the `sys.exit(n)` function call stops the program, and returns the value of the integer `n` to the operating system as an error code.

An alternative solution (compare with the exception hierarchy on slide 104):

```
import sys
try:
    f = open('myfilename.txt', 'r')
except OSError as error:
    print("The file couldn't be opened. " +
          "This program stops here.")
    print("Details: {}".format(error))
    sys.exit(1)         #a way to exit the program

for line in f:
    print(line, end='')
f.close()
```

which produces

```
The file couldn't be opened. This program stops here.
Details: [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*".)

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.

Raising exceptions example

```
>>> 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 coding*:

```
def my_complicated_function(x):
    message = "Called with x={}".format(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
```

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
```

LAB5

Printing

Printing basics

- 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
```

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

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

- 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
>>>
```

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: 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

```
>>> 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'
```

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

```
>>> "%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

```
>>> '%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'
```

There is also the format specifier `%s` that expects a string, or an object that can provide its own string representation.

Combined with a width specifier, this can be used to align columns of strings in tables:

```
>>> "%10s" % "apple"
'     apple'
>>> "%10s" % "banana"
'    banana'
```

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
```

Very similar syntax exists in C and Matlab, amongst others for formatted data output to screen and files.

New style string formatting (`format` method)

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

Basic ideas in examples:

- Pairs of curly braces are the placeholders.

```
>>> "{} needs {} pints".format('Peter', 4)
'Peter needs 4 pints'
```

- Can *index* into the arguments given to `format`:

```
>>> "{0} needs {1} pints".format('Peter',4)
'Peter needs 4 pints'
>>> "{1} needs {0} pints".format('Peter',4)
'4 needs Peter pints'
```

- We can refer to objects through a name:

```
>>> "{name} needs {number} pints".format(\
...     name='Peter',number=4)
'Peter needs 4 pints'
```

- 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.'
```

This is a powerful and elegant way of string formatting.

Further Reading

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

What formatting should I use?

- The `.format` method most elegant and versatile
- `%` operator style okay, links to Matlab, C, ...
- Choice partly a matter of taste
- Should be aware (in a passive sense) of different possible styles (so we can read code from others)

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.)

The `str` function and `__str__` method

All objects in Python should provide a method `__str__` which returns a nice 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'
```

The `str` function is extremely convenient as it allows us to print more complicated objects, such as a list

```
>>> b = [3, 4.2, ['apple', 'banana'], (0, 1)]
>>> str(b)
"[3, 4.2, ['apple', 'banana'], (0, 1)]"
```

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

- use the "%s" format specifier in %-operator formatting to print `x`
- use the "`{}`" format specifier in `.format` to print `x`
- pass the object `x` directly to the print command

```
>>> b = [3, 4.2, ['apple', 'banana'], (0, 1)]
>>> b.__str__()
"[3, 4.2, ['apple', 'banana'], (0, 1)]"
>>> str(b)
"[3, 4.2, ['apple', 'banana'], (0, 1)]"
>>> "%s" % b
"[3, 4.2, ['apple', 'banana'], (0, 1)]"
>>> "{}".format(b)
"[3, 4.2, ['apple', 'banana'], (0, 1)]"
>>> print(b)
[3, 4.2, ['apple', 'banana'], (0, 1)]
```

The `repr` function and `__repr__` method

- The **`repr`** function should convert a given object into an *as accurate as possible* string representation
- The **`str`** function, in contrast, aims to return an "informal" representation of the object that is useful to humans.
- The **`repr`** function will generally provide a more detailed string than **`str`**.
- Applying **`repr`** to the object `x` will attempt to call `x.__repr__()`.

Example:

```
>>> import datetime
>>> t = datetime.datetime.now()    # current date and time
>>> str(t)
'2016-09-08 14:28:48.648192'       # inofficial representation
                                   # (nice for humans)
>>> repr(t)                        # official representation
'datetime.datetime(2016, 9, 8, 14, 28, 48, 648192)'
```

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 `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
```

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 $f(x) = \sin(x)$

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):
    for i in range(6):
        x = i * 0.5
        print("{} {}".format(x, f(x)))

def square(x):
    return x ** 2

print_f_table(square)
```

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
        print("{} {}".format(x, f(x)))

def square(x):
    return x ** 2

def cubic(x):
    return x ** 3
```

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

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
```

Functions are first class objects

- Functions are *first class objects* $\leftrightarrow$ functions can be given to other functions as arguments
- Example (trigtable.py):

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

produces

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

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”)
- Every Python file can be imported as a module.
- If this module file has a main program, then this is executed when the file is imported. This can be desired but sometimes it is not.
- We describe how a main program can be written which is only executed if the file is run on its own but not if is imported as a library.

The internal `__name__` variable (1)

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

```
def someusefulfunction():
    pass

print("My name is {}".format(__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.

- 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
  ```

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

The internal `__name__` variable (2)

- In summary
  - `__name__` is `"__main__"` if the module file is run on its own
  - `__name__` is the name (type string) of the module if the module file is imported.
- We can therefore use the following `if` statement in `module1.py` to write code that is *only run* when the module is executed on its own:

```
def someusefulfunction():
    pass

if __name__ == "__main__":
    print("I am running on my own.")
```

- This is useful to keep test programs or demonstrations of the abilities of a library module in this “conditional” main program.

Library file example

```
def useful_function():
    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
```

Default arguments

Default argument values

- 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("The area is {}.".format(area(3, 1)))
print("The area is {}.".format(area(2.5, 1)))
print("The area is {}.".format(area(2.5, 2)))
```

- 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("the area is {}".format(area(3)))
print("the area is {}".format(area(2.5)))
print("the area is {}".format(area(2.5, 2)))
```

- If a default value is defined, then this parameter (here `b`) is optional when the function is called.
- Default parameters have to be at the end of the argument list in the function definition.

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("a = {}, b = {}, c = {}"
          .format(a, b, c))


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

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 don't need to know the *order* of the arguments. (This is good.)

Combining keyword arguments with default argument values

- Can combine default value arguments and keyword arguments
- Example: Imagine for a numerical integration routine we use 100 subdivisions unless the user provides a number

```
def trapez(function, a, b, subdivisions=100):
    #code missing here
    pass

import math
int1 = trapez(a=0, b=10, function=math.sin)
int2 = trapez(b=0, function=math.exp,
              subdivisions=1000, a=-0.5)
```

- Note that choosing meaningful variable names in the function definition makes the function more user friendly.
- You may have met default arguments in use before, for example
  - the `open` function uses `mode='r'` as a default value
  - the `print` function uses `end='\n'` as a default value

LAB6

Namespaces

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* functions
- The same variable *name* can be used in a function and in the main program but they can refer to different objects and do not interfere:

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

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

which produces this output

```
I am local
I am global
```

- Imported modules have their own name space.

So global and local variables can't see each other?

- not quite. Let's read the small print:
  - If — within a function – we try to access an object through its name, then Python will look for this name
    - first in the local name space (*i.e.* within that function)
    - then in the global name space

    If the variable can not be found, then a `NameError` is raised.

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

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

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

Output:

```
I am global
```

- 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:

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

- To *modify* global variables within a local namespace, we need to use the `global` keyword.

  (This is not recommended so we won't explain it. See also next slide.)

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 modules which is good engineering practice and essential to control complexity of software.
- 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)
- 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.

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.

Python IDEs

Integrated DeveLopment Environment: IDLE

- IDLE http://en.wikipedia.org/wiki/IDLE_(Python)__ (comes with Python)
- two windows: program and python prompt
- F5 to execute Python program
- simple
- portable (written in Python)

IPython (interactive python)

- Interactive Python (`ipython` from Command Prompt/Unix-shell)
- command history (across sessions), auto completion,
- special commands:
  - `%run test` will execute file `test.py` in current name space (in contrast to IDLE this does not remove all existing objects from global 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
- Much (!) more (read at http://ipython.org)

IPython's QT console

- Prompt as IPython (with all it's features): running in a *graphics console* rather than in *text console*
- can inline matplotlib figures
- Read more at http://ipython.org/ipython-doc/dev/interactive/qtconsole.html

Jupyter Notebook

- Used to be the IPython Notebook, but now supports many more languages than Python, thus a new name was chosen.
- Jupyter Notebook creates an *executable* document that is hosted in a web browser.
- We recommend you try this at some point, it has great value for computational engineering and research.
- Read more at http://jupyter.org

... and many others

Including

- Eclipse
- Spyder
- PyCharm (commercial)
- Emacs
- vi, vim
- Sublime Text (commercial)
- Kate

List comprehension

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(10)]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]

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

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

Can be useful to populate lists with numbers quickly

- Example 1:

```
>>> xs = [i for i in range(10)]
>>> xs
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> ys = [x ** 2 for x in xs]
>>> ys
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Example 2:

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

- Example 3

```
>>> words = 'The quick brown fox jumps \
... over the lazy dog'.split()
>>> print words
['The', 'quick', 'brown', 'fox', 'jumps',
 'over', 'the', 'lazy', 'dog']
```

```
>>> stuff = [[w.upper(), w.lower(), len(w)]
             for w in words]
>>> for i in stuff:
...     print(i)
...
['THE', 'the', 3]
['QUICK', 'quick', 5]
['BROWN', 'brown', 5]
['FOX', 'fox', 3]
['JUMPS', 'jumps', 5
['OVER', 'over', 4]
['THE', 'the', 3]
['LAZY', 'lazy', 4]
['DOG', 'dog', 3]
```

List comprehension with conditional

- Can extend list comprehension syntax with `if CONDITION` to include only elements for which `CONDITION` 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]
```

Dictionaries

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*.
- 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'
```

- `d.keys()` returns a list of all keys:

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

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

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

Dictionary example 1

```
order = {}          # create empty dictionary

# add orders as they come in
order['Peter'] = 'Pint of bitter'
order['Paul'] = 'Half pint of Hoegarden'
order['Mary'] = 'Gin Tonic'

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

which produces this output:

```
Paul requests Half pint of Hoegarden
Peter requests Pint of bitter
Mary requests Gin Tonic
```

Dictionary

Some more technicalities:

- The dictionary key can be any (immutable) Python object. This includes:
  - numbers
  - strings
  - tuples.
- dictionaries are very fast in retrieving values (when given the key)

Dictionary use case

- What are dictionnaries good for? Consider this example:

```
dic = {}
dic["Andy C"] = "room 1031"
dic["Ken"]    = "room 1027"
dic["Hans"]   = "room 1033"

for key in dic.keys():
    print("{} works in {}"
          .format(key, dic[key]))
```

Output:

```
Hans works in room 1033
Andy C works in room 1031
Ken works in room 1027
```

- Without dictionary:

```
people = ["Hans", "Andy C", "Ken"]
rooms  = ["room 1033", "room 1031", \
     "room 1027"]

# possible inconsistency here since we have
# two lists
if not len(people) == len(rooms):
    raise ValueError("people and rooms " +
                     "differ in length")

for i in range(len(rooms)):
    print ("{} works in {}".format(people[i],
                                   rooms[i])
```

Iterating over dictionaries

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

```
order = {}          # create empty dictionary

order['Peter'] = 'Pint of bitter'
order['Paul'] = 'Half pint of Hoegarden'
order['Mary'] = 'Gin Tonic'

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

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

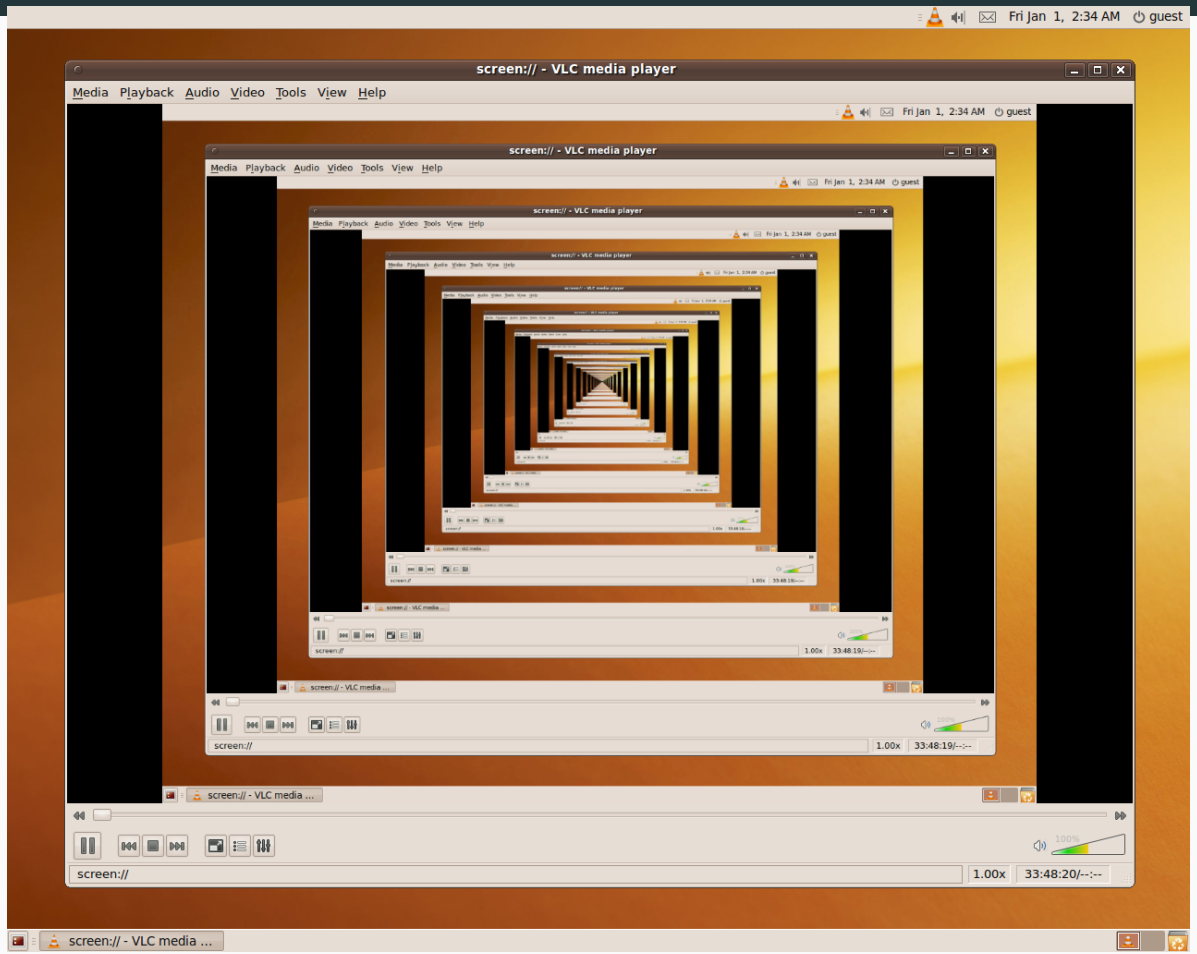
Summary dictionaries

What to remember:

- Python provides dictionaries
- very powerful construct
- a bit like a data base (and values can be dictionary objects)
- fast to retrieve value
- likely to be 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)

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 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 197:
   - 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.)

Common Computational Tasks

Overview common computational tasks

- Data file processing, python & `numpy` (`array`)
- Random number generation and fourier transforms (`numpy`)
- Linear algebra (`numpy`)
- Interpolation of data (`scipy.interpolation.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.odeint`)

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

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?

- Often need to know when a particular function reaches a value, for example the water temperature $T(t)$ reaching $373\,K$. In that case, we define

  $$f(t) = T(t) - 373$$

  and search the root t_0 for $f(t)$

We introduce two methods:

- Bisection method
- Newton method

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 $f_{\text{tol}} = 1e-6$

The bisection method returns x so that $|f(x)| <$ftol

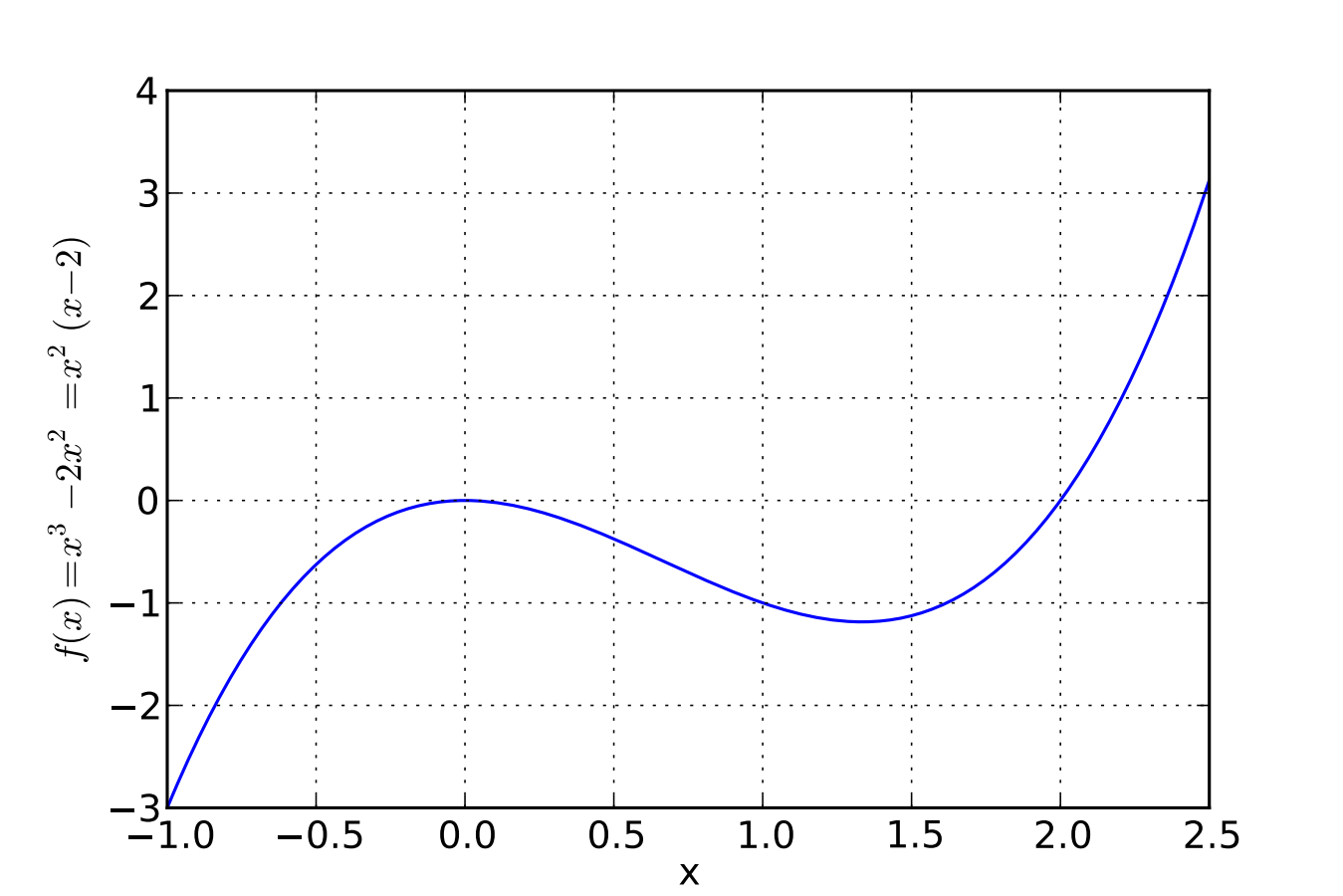
1. $x = (a + b)/2$
2. while $|f(x)| >$ ftol do
   - if $f(x)f(a) > 0$
     then $a \leftarrow x$ # throw away left half
     else $b \leftarrow x$ # throw away right half
   - $x = (a + b)/2$
3. return x

The bisection 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 $|\texttt{b} - \texttt{a}| < \texttt{xtol}$.

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 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("Root x is approx. x={:14.12g}.".format(x))
print("The error is less than 1e-6.")
print("The exact error is {}.".format(2 - x))
```

produces

```
Root x is approx. x= 2.00000023842.
The error is less than 1e-6.
The exact error is -2.38418579101562e-07.
```

The Newton method

- Newton method for root finding: find x_0 so that $f(x_0) = 0$.
- Idea: close to the root, the tangent of $f(x)$ is likely to point to the root. Make use of this information.
- Algorithm:
  while $|f(x)| >$ftol, do

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

  where $f'(x) = \frac{\mathrm{d}f}{\mathrm{d}x}(x)$.
- Much better convergence than bisection method
- but not guaranteed to converge.
- Need a good initial guess x for the root.

Using Newton algorithm from `scipy`

```
from scipy.optimize import newton

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 = newton(f, x0=1.6)

print("Root x is approx. x={:14.12g}.".format(x))
print("The error is less than 1e-6.")
print("The exact error is {}.".format(2 - x))
```

produces

```
Root x is approx. x=              2.
The error is less than 1e-6.
The exact error is 9.769962616701378e-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`

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 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("Root x is approx. x={:14.12g}.".format(x))
print("The error is less than 1e-6.")
print("The exact error is {}.".format(2 - x))
```

produces:

```
Root x is approx. x= 2.00000001896.
The error is less than 1e-6.
The exact error is -1.8958284962285938e-08.
```

Using fsolve algorithm from `scipy`

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

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

x = fsolve(f, x0=[1.6])     # main program starts here

print("Root x is approx. x={}.".format(x))
print("The error is less than 1e-6.")
print("The exact error is {}.".format(2 - x[0]))
```

produces:

```
Root x is approx. x=[ 2.].
The error is less than 1e-6.
The exact error is 0.0.
```

Derivatives

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.
- The following slides show
  - the forward difference technique
  - the backward difference technique and the

- central difference technique to approximate the derivative of a function.
- We also derive the accuracy of each of these methods.

The 1st derivative

- (Possible) Definition of the derivative (or "*differential* operator" $\frac{\mathrm{d}}{\mathrm{d}x}$)

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

- Use *difference* operator to approximate differential operator

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

- $\Rightarrow$ can now compute *an approximation* of f' simply by evaluating f.
- This is called the *forward difference* because we use $f(x)$ and $f(x+h)$.
- Important question: How accurate is this approximation?

Accuracy of the forward difference

- Formal derivation using the Taylor series of f around x

$$\begin{aligned} f(x+h) &= \sum_{n=0}^{\infty} h^n \frac{f^{(n)}(x)}{n!} \\ &= f(x) + hf'(x) + h^2\frac{f''(x)}{2!} + h^3\frac{f'''(x)}{3!} + \dots \end{aligned}$$

- Rearranging for $f'(x)$

$$hf'(x) = f(x+h) - f(x) - h^2\frac{f''(x)}{2!} - h^3\frac{f'''(x)}{3!} - \dots$$

$$\begin{aligned} f'(x) &= \frac{1}{h}\left(f(x+h) - f(x) - h^2\frac{f''(x)}{2!} - h^3\frac{f'''(x)}{3!} - \dots\right) \\ &= \frac{f(x+h)-f(x)}{h} - \frac{h^2\frac{f''(x)}{2!} - h^3\frac{f'''(x)}{3!}}{h} - \dots \\ &= \frac{f(x+h)-f(x)}{h} - h\frac{f''(x)}{2!} - h^2\frac{f'''(x)}{3!} - \dots \end{aligned}$$

Accuracy of the forward difference (2)

$$f'(x) = \frac{f(x+h)-f(x)}{h} - \underbrace{h\frac{f''(x)}{2!} - h^2\frac{f'''(x)}{3!} - \dots}_{E_{\text{forw}}(h)}$$

$$f'(x) = \frac{f(x+h)-f(x)}{h} + E_{\text{forw}}(h)$$

- Therefore, the error term $E_{\text{forw}}(h)$ is

$$E_{\text{forw}}(h) = -h\frac{f''(x)}{2!} - h^2\frac{f'''(x)}{3!} - \dots$$

- Can also be expressed as

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

The 1st derivative using the backward difference

- Another definition of the derivative (or "differential operator" $\frac{\mathrm{d}}{\mathrm{d}x}$)

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

- Use difference operator to approximate differential operator

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

- This is called the *backward difference* because we use $f(x)$ and $f(x-h)$.
- How accurate is the backward difference?

Accuracy of the backward difference

- Formal derivation using the Taylor Series of f around x

$$f(x-h) = f(x) - hf'(x) + h^2\frac{f''(x)}{2!} - h^3\frac{f'''(x)}{3!} + \dots$$

- Rearranging for $f'(x)$

$$hf'(x) = f(x) - f(x-h) + h^2\frac{f''(x)}{2!} - h^3\frac{f'''(x)}{3!} - \dots$$

$$\begin{aligned} f'(x) &= \frac{1}{h}\left(f(x) - f(x-h) + h^2\frac{f''(x)}{2!} - h^3\frac{f'''(x)}{3!} - \dots\right) \\ &= \frac{f(x) - f(x-h)}{h} + \frac{h^2\frac{f''(x)}{2!} - h^3\frac{f'''(x)}{3!}}{h} - \dots \\ &= \frac{f(x) - f(x-h)}{h} + h\frac{f''(x)}{2!} - h^2\frac{f'''(x)}{3!} - \dots \end{aligned}$$

Accuracy of the backward difference (2)

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \underbrace{h\frac{f''(x)}{2!} - h^2\frac{f'''(x)}{3!} - \dots}_{E_{\text{back}}(h)}$$

$$f'(x) = \frac{f(x) - f(x-h)}{h} + E_{\text{back}}(h) \tag{7}$$

- Therefore, the error term $E_{\text{back}}(h)$ is

$$E_{\text{back}}(h) = h\frac{f''(x)}{2!} - h^2\frac{f'''(x)}{3!} - \dots$$

- Can also be expressed as

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

Combining backward and forward differences (1)

The approximations are

- forward:

$$f'(x) = \frac{f(x+h) - f(x)}{h} + E_{\text{forw}}(h) \tag{8}$$

- backward

$$f'(x) = \frac{f(x) - f(x-h)}{h} + E_{\text{back}}(h) \tag{9}$$

$$\begin{aligned} E_{\text{forw}}(h) &= -h\frac{f''(x)}{2!} - h^2\frac{f'''(x)}{3!} - h^3\frac{f''''(x)}{4!} - h^4\frac{f'''''(x)}{5!} - \dots \\ E_{\text{back}}(h) &= h\frac{f''(x)}{2!} - h^2\frac{f'''(x)}{3!} + h^3\frac{f''''(x)}{4!} - h^4\frac{f'''''(x)}{5!} + \dots \end{aligned}$$

$\Rightarrow$ Add equations (8) and (9) together, then the error cancels partly.

Combining backward and forward differences (2)

Add these lines together

$$\begin{aligned} f'(x) &= \frac{f(x+h)-f(x)}{h} + E_{\text{forw}}(h) \\ f'(x) &= \frac{f(x)-f(x-h)}{h} + E_{\text{back}}(h) \end{aligned}$$

$$2f'(x) = \frac{f(x+h)-f(x-h)}{h} + E_{\text{forw}}(h) + E_{\text{back}}(h)$$

Adding the error terms:

$$E_{\text{forw}}(h) + E_{\text{back}}(h) = -2h^2\frac{f'''(x)}{3!} - 2h^4\frac{f'''''(x)}{5!} - \dots$$

The combined (central) difference operator is

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

with

$$E_{\text{cent}}(h) = -h^2\frac{f'''(x)}{3!} - h^4\frac{f'''''(x)}{5!} - \dots$$

Central difference

- Can be derived (as on previous slides) by adding forward and backward difference
- Can also be interpreted geometrically by defining the differential operator as

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

  and taking the finite difference form

$$\frac{\mathrm{d}f}{\mathrm{d}x}(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

- Error of the central difference is only $\mathcal{O}(h^2)$, *i.e.* better than forward or backward difference

It is generally the case that symmetric differences are more accurate than asymmetric expressions.

Example (1)

Using forward difference to estimate the derivative of $f(x) = \exp(x)$

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

Since we compute the difference using values of f at x and $x + h$, it is natural to interpret the numerical derivative to be taken at $x + \frac{h}{2}$:

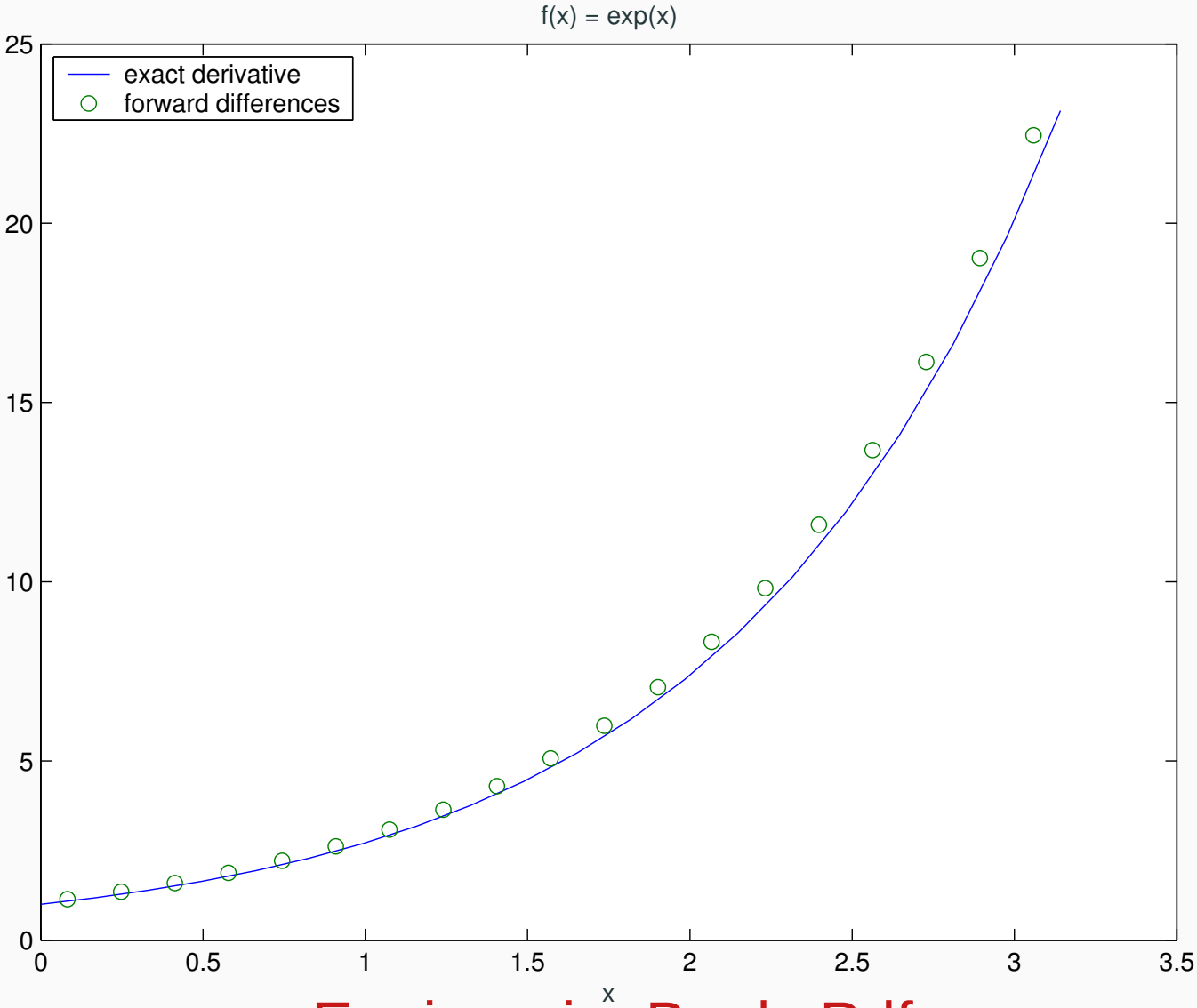
$$f'\left(x + \frac{h}{2}\right) \approx \frac{f(x+h) - f(x)}{h} = \frac{\exp(x+h) - \exp(x)}{h}$$

Numerical example:

- $h = 0.1, x = 1$
- $f'(1.05) \approx \frac{\exp(1.1) - \exp(1)}{0.1} = 2.8588$
- Exact answers is $f'(1.05) = \exp(1.05) = 2.8577$

Example (2)

Comparison: forward difference and exact derivative of $\exp(x)$



Summary

- Can approximate derivatives of f numerically
- need only function evaluations of f
- three different difference methods

name	formula	error
forward	$f'(x) = \frac{f(x+h)-f(x)}{h}$	$\mathcal{O}(h)$
backward	$f'(x) = \frac{f(x)-f(x-h)}{h}$	$\mathcal{O}(h)$
central	$f'(x) = \frac{f(x+h)-f(x-h)}{2h}$	$\mathcal{O}(h^2)$

- central difference is most accurate
- Euler's method (ODE) can be derived from forward difference
- Newton's root finding method can be derived from forward difference

LAB7

Note: Euler's (integration) method — derivation using finite difference operator

- Use forward difference operator to approximate differential operator

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

- Change differential to difference operator in $\frac{\mathrm{d}y}{\mathrm{d}x} = f(x, y)$

$$\begin{aligned} f(x,y) = \frac{\mathrm{d}y}{\mathrm{d}x} &\approx \frac{y(x+h) - y(x)}{h} \\ hf(x,y) &\approx y(x+h) - y(x) \\ \implies y_{i+1} &= y_i + hf(x_i, y_i) \end{aligned}$$

- $\Rightarrow$ Euler's method (for ODEs) can be derived from the forward difference operator.

Note: Newton's (root finding) method — derivation from Taylor series

- We are looking for a root, *i.e.* we are looking for a x so that $f(x) = 0$.
- We have an initial guess x_0 which we refine in subsequent iterations:

$$x_{i+1} = x_i - h_i \quad \text{where} \quad h_i = \frac{f(x_i)}{f'(x_i)}. \tag{10}$$

.

- This equation can be derived from the Taylor series of f around x. Suppose we guess the root to be at x and $x + h$ is the actual location of the root (so h is unknown and $f(x + h) = 0$):

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + \dots \\ 0 &= f(x) + hf'(x) + \dots \\ \implies 0 &\approx f(x) + hf'(x) \\ \iff h &\approx -\frac{f(x)}{f'(x)}. \end{aligned} \tag{11}$$

Numpy

numpy

- is an interface to high performance linear algebra libraries (ATLAS, LAPACK, BLAS)
- provides
  - the `array` object
  - fast mathematical operations over arrays
  - linear algebra, Fourier transforms, Random Number generation
- Numpy is *not* part of the Python standard library.

numpy arrays (vectors)

- An array is a sequence of objects
- all objects in one array are of the same type

Here are a few examples:

```
>>> import numpy as np
>>> a = np.array([1, 4, 10])
>>> type(a)
<class numpy.ndarray>
>>> a.shape
(3,)
>>> a ** 2
array([  1,  16, 100])
>>> np.sqrt(a)
array([ 1.        ,  2.        ,  3.16227766])
>>> a > 3
array([False,  True,  True], dtype=bool)
```

Array creation

Can create from other sequences through **array** function:

- 1d-array (vector)

```
>>> from numpy import array
>>> a = array([1, 4, 10])
>>> a
array([ 1,  4, 10])
>>> print(a)
[ 1  4 10]
```

- 2d-array (matrix):

```
>>> B = array([[0, 1.5], [10, 12]])
>>> B
array([[  0. ,   1.5],
       [ 10. ,  12. ]])
>>> print(B)
[[  0.    1.5]
 [ 10.   12. ]]
```

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.

Example:

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

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. ])
```

Array size

The total number of elements is given through the `size` attribute:

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

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

```
>>> a.nbytes
12
>>> B.nbytes
32
```

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 the create the array, for example:

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

Important array types

- For numerical calculations, we normally use double floats which are knows as *float64* or short `float` in this text.:

```
>>> a2 = array([1, 4, 10], numpy.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 creation II

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

```
>>> numpy.zeros((3, 3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> numpy.zeros((4,))          # (4,) is tuple
array([ 0.,  0.,  0.,  0.])
>>> numpy.zeros(4)             # works as well,
                               # although 4 is
                               # not tuple.
array([ 0.,  0.,  0.,  0.])

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

Array indexing (1d arrays)

```
>>> x = numpy.array(range(0, 10, 2))
>>> x
array([0, 2, 4, 6, 8])
>>> x[3]
6
>>> x[4]
8
>>> x[-1]
```

Can query length as for any sequence:

```
>>> len(x)
5
>>> x.shape
(5,)                # <=> length of 1d array is 5
```

Array indexing (2d arrays)

```
>>> C = numpy.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]
0
>>> C[2, 0]
8
>>> C[2, -1]
11
>>> C[-1, -1]
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:

```
>>> y = numpy.arange(10)
>>> y
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> y[0:5]                # slicing as we know it
array([0, 1, 2, 3, 4])
>>> y[0:5:1]              # slicing with index 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]              # positive index step size
array([0, 1, 2, 3, 4])
>>> y[0:5:-1]             # negative index step size
array([], dtype=int64)
>>> y[5:0:-1]             # from end to front
array([5, 4, 3, 2, 1])
>>> y[5:0:-2]             # in steps of two
array([5, 3, 1])
>>> y[::-1]               # reverses array elements
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

Creating copies of arrays

Create copy of 1d array:

```
>>> copy_y = y[:]
```

Could also use `copy_y = y[::]` to create a copy.

To create a copy with reverse order of elements, we can use:

```
>>> y[::-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

To create new array `z` of the same size as `y` (filled with zeros, say) we can use (for example):

```
>>> y
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> z = numpy.zeros(y.shape)
>>> z
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

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])
```

Other linear algebra tools

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

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

Curve fitting

- We typically fit lower order polynomials or other functions (which are the model that we expect the data to follow) through a number of points (often measurements).
- We typically have many more points than degrees of freedom, and would employ techniques such as least squared fitting.
- The function `numpy.polyfit` provides this functionality for polynomials.
- The function `scipy.optimize.curve_fit` provides curve fitting for generic functions (not restricted to polynomials).

Solving linear systems of equations

- `numpy.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 $\mathbf{x}$ as an array object:

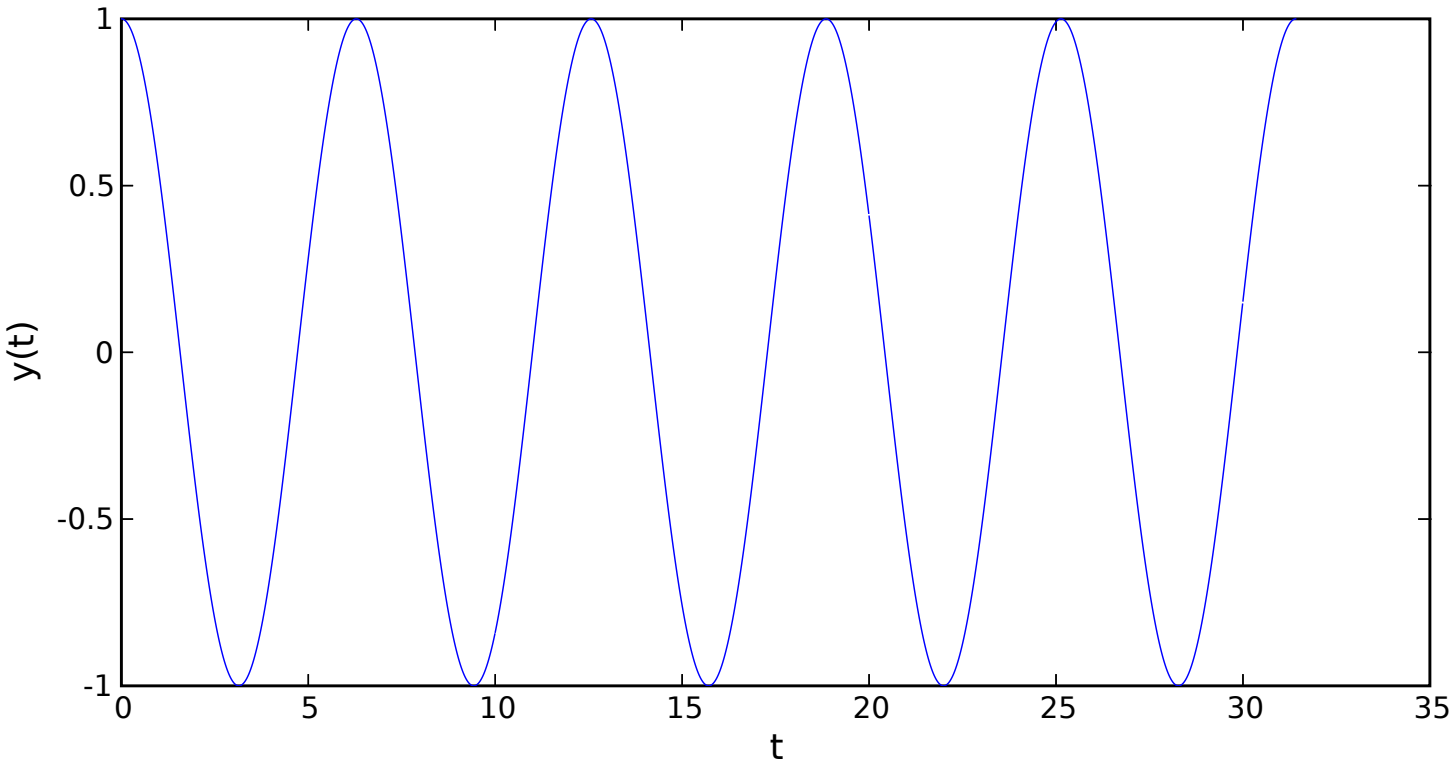
```
>>> A = numpy.array([[1, 0], [0, 2]])
>>> b = numpy.array([1, 4])
>>> from numpy import linalg as LA
>>> x = LA.solve(A, b)
>>> x
array([ 1.,  2.])
>>> numpy.dot(A, x)        # Computing A*x
array([ 1.,  4.])          # this should be b
```

Other comments

- numpy provides fast array operations (comparable to Matlab's matrices)
- fast if number of elements is large: for an array with one element, `numpy.sqrt` will be slower than `math.sqrt`
- speed-ups of up to factor 50 to 300 are possible using numpy instead of lists
- Consult Numpy documentation if used outside this course.
- Matlab users may want to read Numpy for Matlab Users

Plotting arrays (vectors)

```
import pylab
import numpy as N

t = N.arange(0, 10 * N.pi, 0.01)
y = N.cos(t)

pylab.plot(t, y)
pylab.xlabel('t')
pylab.ylabel('y(t)')
pylab.show()
```



Matplotlib / Pylab

- 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

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 when the `matplotlib.show()` command is used. We can manipulate the view interactively in that window.
- Within the notebook, you can use `%matplotlib notebook` which embeds an interactive window in the note book.

Pylab

Pylab is a Matlab-like (state-driven) plotting interface (and comes with matplotlib).

- Convenient for 'simple' plots
- Check examples in lecture note text book and
- Make use of `help(pylab.plot)` to remind you of line styles, symbols etc.
- Check gallery at http://matplotlib.org/gallery.html#pylab_examples

Matplotlib.pyplot

Matplotlib.pyplot is an object oriented plotting interface.

- Very fine grained control over plots
- Check gallery at Matplotlib gallery
- Try Matplotlib notebook (on module's home page) as an introduction and useful reference.

LAB8

Higher Order Functions 2: Functional tools

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

```
>>> 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 * x

y, abserr = quad(f, a=0, b=2)
print("Int f(x)=x^2 from 0 to 2 = {:f} +- {:g}"
      .format(y, abserr))
```

- With lambda (`lambda1b.py`):

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

Both programs produce the same output:

```
Int f(x)=x^2 from 0 to 2 = 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, sequence)` → `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` is often useful in `map`:

```
>>> 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.
- Example:

```
>>> c = "The quick brown fox jumps".split()
>>> print(c)
['The', 'quick', 'brown', 'fox', 'jumps']
>>> def len_gr_4(s):
...     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']
```

Equivalent operation using list comprehension:

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

Examples filter

- Example:

```
>>> def is_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:

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

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("Called with x={}, y={}".format(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(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:

```
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).

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”
- `map` transforms a sequence to another sequence (of same length)
- `filter` filters a sequence (reduces number of elements)
- `reduce` carries out an operation that ”collects” information (sum, product, ...), for example reducing the sequence to a single number.
- `list comprehension` transforms a list (can include filtering).
- Hint: if you need to use a `lambda` in a `map`, you are probably better off using list comprehension.

Standard example: squaring elements in list

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]
```

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
```

Object Orientation and all that

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("waiting customer: {}".format(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

Improved FIFO solution (`fifoqueue2.py`)

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

def show(queue):
    for name in queue:
        print("waiting customer: {}".format(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("{} customers in queue1:".format(length(q1)));  show(q1)
print("{} customers in queue2:".format(length(q2)));  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("waiting customer: {}".format(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("{} customers in queue1:".format(q1.length())); 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("waiting customer: {}".format(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("{} customers in queue1:".format(q1_length()); q1_show()
print("{} customers in queue2:".format(q2_length()); q2_show()
```

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 32 pp)

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).

Often written as

$$I = \int_a^b f(x)\mathrm{d}x \tag{12}$$

- 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

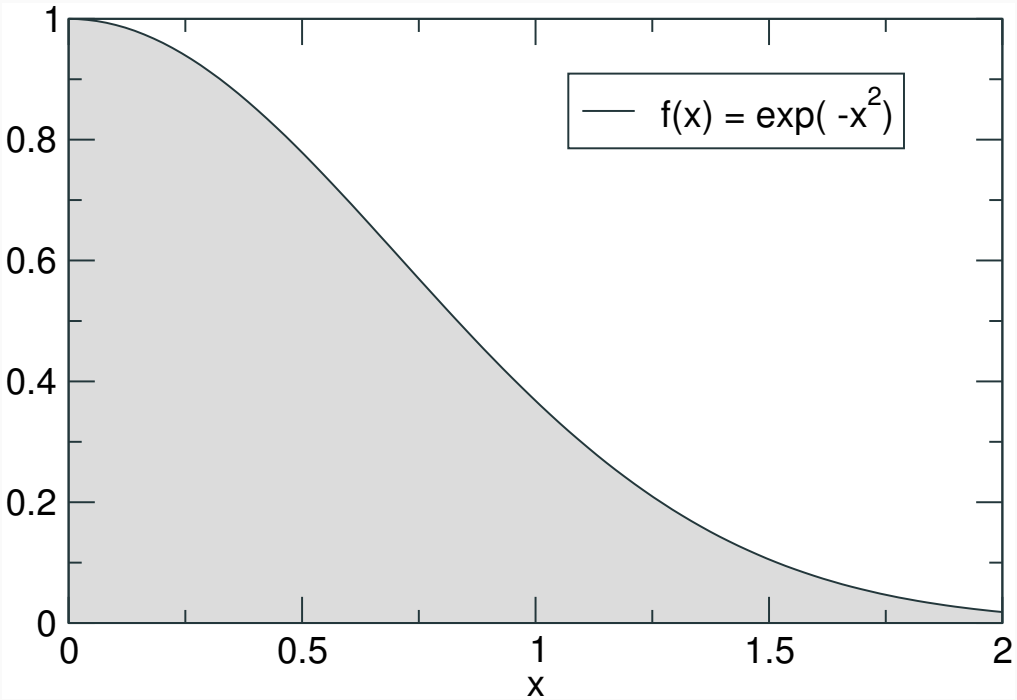
$$y' \equiv \frac{\mathrm{d}y}{\mathrm{d}x} = f(x, y) \tag{13}$$

- example: $\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{1}{m}(g - cv^2)$
- solution is $y(x): \quad \mathbb{R} \to \mathbb{R}, \; x \mapsto y(x)$ (*i.e.* a function)
- right hand side $f(x, y)$ depends on x and on solution y
- Can write (13) 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

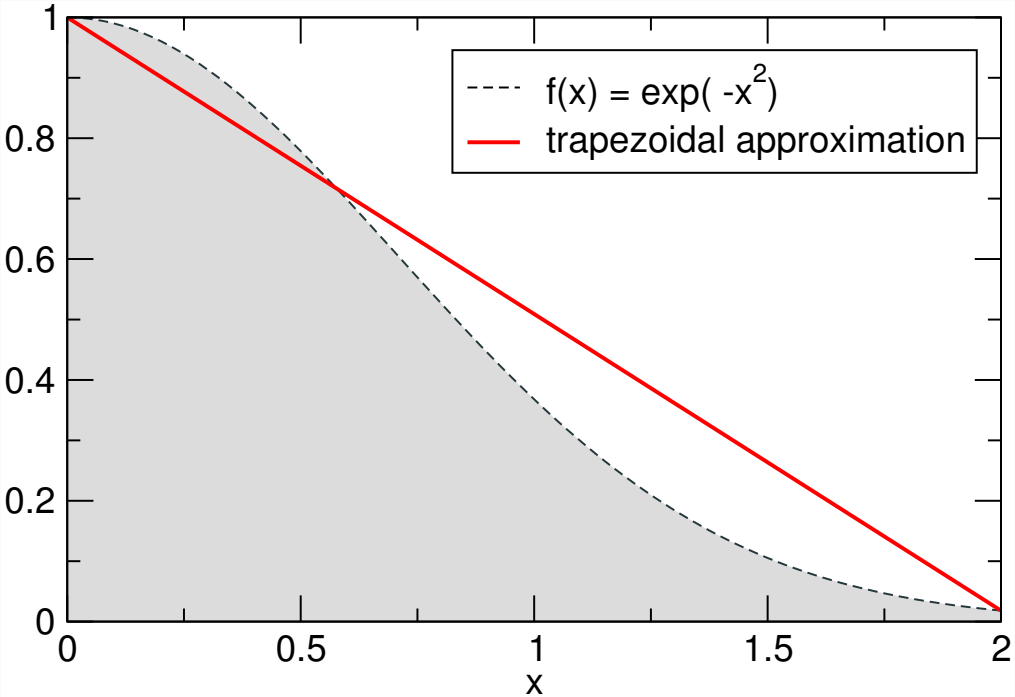
Example:

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



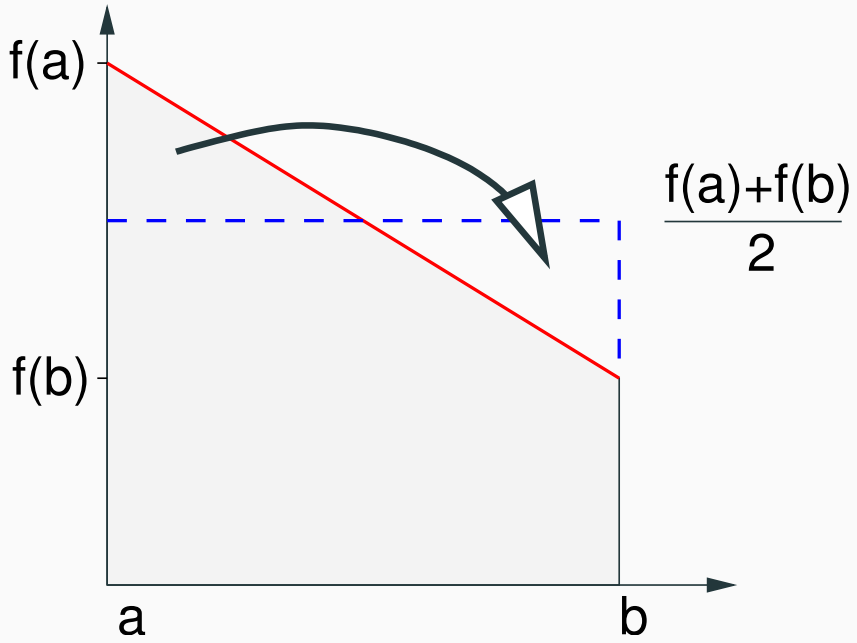
Simple trapezoidal rule

- Approximate function by straight line



Simple trapezoidal rule (2)

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



- Result

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

Simple trapezoidal rule (3)

Aim: compute

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

Strategy:

- approximate $f(x)$ with a linear function $p(x)$:

$$p(x) \approx f(x)$$

- compute the area A underneath that function $p(x)$:

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

- approximate

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

Simple trapezoidal rule (4) Example

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

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

- What is the (correct) analytical answer?
  Integrating polynomials:

$$I = \int_a^b x^k \mathrm{d}x = \left[\frac{1}{k+1} x^{k+1} \right]_a^b$$

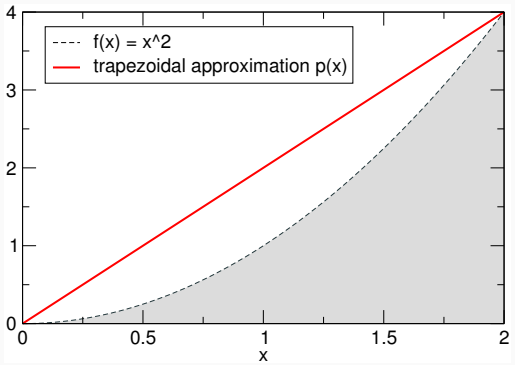
- for $a = 0$ and $b = 2$ and $k = 2$

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

- 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



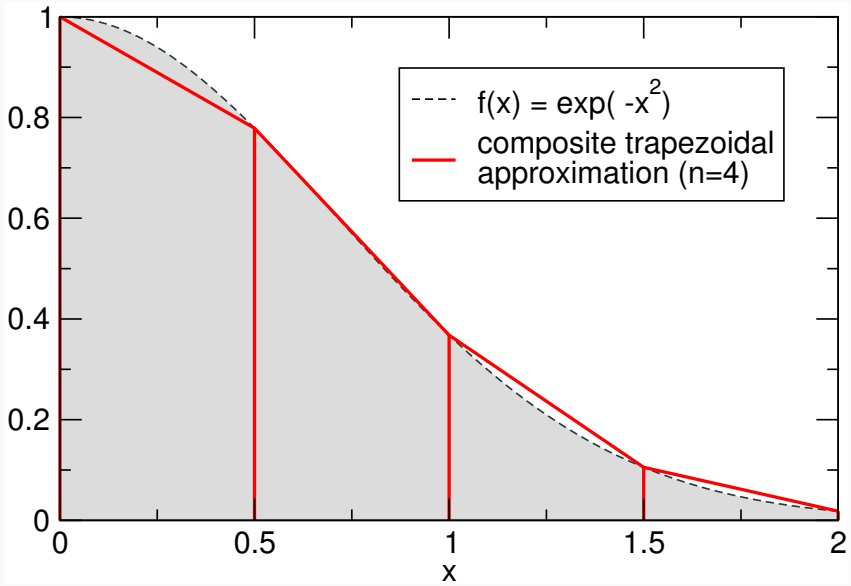
- The linear approximation, $p(x)$, overestimates $f(x)$ everywhere (except at $x = a$ and $x = b$).
  Therefore, the integral of $p(x)$ is greater than the integral of $f(x)$.

  (More formally: $f(x)$ is convex on $[a, b] \iff f''(x) \geq 0 \quad \forall x \in [a, b]$.)

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

One of the important (and difficult) questions in numerical analysis and computing is:

- How accurate is my approximation?

For integration methods, we are interest in 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

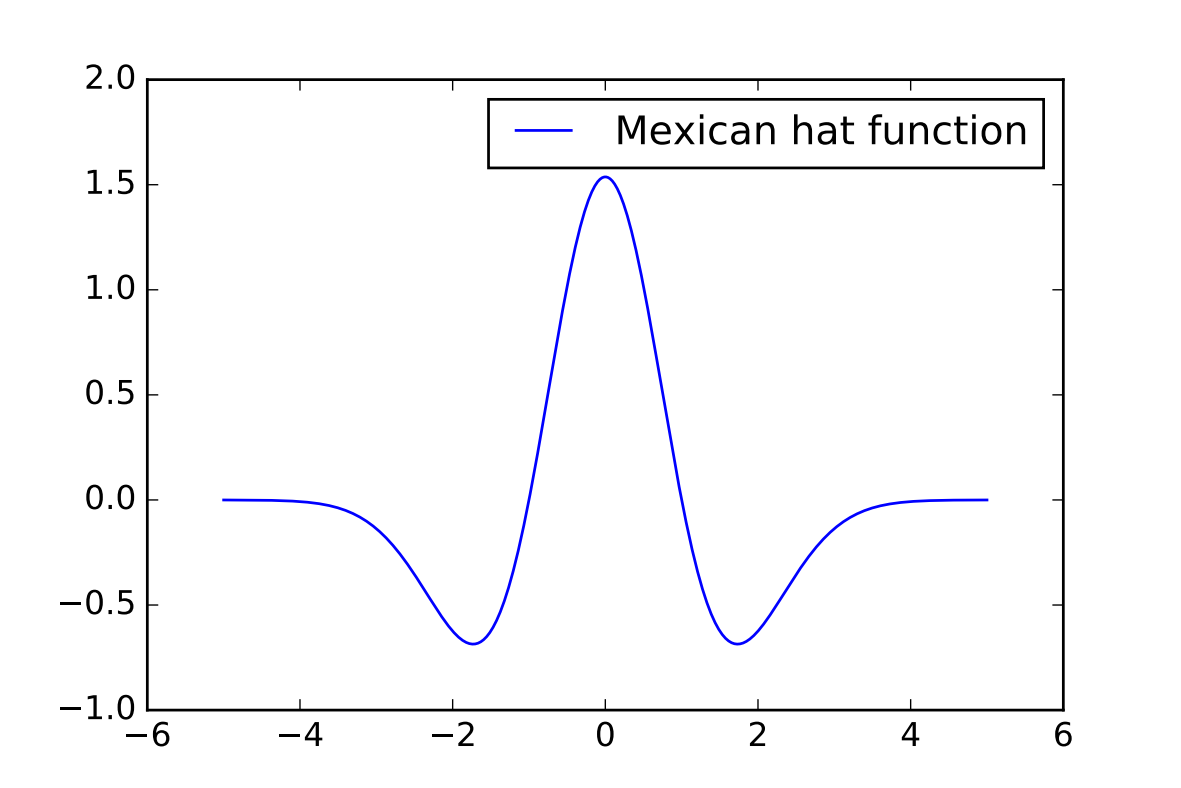
$$I = \int_a^b f(x)\mathrm{d}x$$

- 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.

Numpy usage examples

Making calculations fast 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



Making calculations fast with numpy

```
"""Demo: practical use of numpy (mexhat-numpy.py)"""
import time
import math
import numpy as np

N = 10000

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. / 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, see
    http://en.wikipedia.org/wiki/Mexican_hat_wavelet for
    equation (13 Dec 2011)"""
    c = 2. / math.sqrt(3 * sigma) * math.pi ** 0.25
    return c * (1 - t ** 2 / sigma ** 2) * \
        np.exp(-t ** 2 / (2 * sigma ** 2))

def test_is_really_the_same():
    """Checking whether mexhat_np and mexhat_py produce
    the same results."""
    xs1, ys1 = loop1()
    xs2, ys2 = loop2()
    deviation = math.sqrt(sum((ys1 - ys2) ** 2))
    print("error:", deviation)
    assert deviation < 1e-15
```

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

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

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

```
    f()
    stoptime = time.time()
    return stoptime - starttime

def make_plot(filenameroot):
    import pylab
    pylab.figure(figsize=(6, 4))
    xs, ys = loop2()
    pylab.plot(xs, ys, label='Mexican hat function')
    pylab.legend()
    pylab.savefig(filenameroot + '.png')
    pylab.savefig(filenameroot + '.pdf')

def main():
    test_is_really_the_same()
    make_plot('mexhat1d')
    time1 = time_this(loop1)
    time2 = time_this(loop2)
```

```
    print("Numpy version is %.1f times faster"
          % (time1 / time2))

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

Produces this output:

```
error: 2.220296320536979e-16
Numpy version is 109.6 times faster
```

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 same timing information.

Array objects of shape () behave like scalars

```
>>> import numpy as np
>>> np.sqrt(4.)          # apply numpy-sqrt to scalar
2.0                      # looks like float
>>> type(np.sqrt(4.))    # but is numpy-float
<class numpy.float64>
>>> float(np.sqrt(4.)) # but can convert to float
2.0
>>> a = np.sqrt(4.)      # what shape is the
                         # numpy-float?
>>> a.shape
()
>>> type(a)              # just to remind us
```

```
<class numpy.float64> # of the type
>>>
```

So numpy-scalars (i.e. arrays with shape `()`) can be converted to float. In fact, this happens implicitly:

```
>>> import numpy as np
>>> import math
>>> math.sqrt(np.sqrt(81))
3.0
```

Conversion to float fails if array has more than one element:

```
>>> import numpy as np
>>> a = np.array([10., 20., 30.])
>>> a
array([ 10.,  20.,  30.])
>>> print(a)
[ 10.  20.  30.]
>>> type(a)
<class numpy.ndarray>
>>> a.shape
(3,)
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only length-1 arrays can be converted
           to Python scalars
>>> math.sqrt(a)
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: only length-1 arrays can be converted
           to Python scalars
```

However, if the array contains only one number, then the conversion is possible:

```
>>> b = np.array(4.0)
>>> type(b)
<class numpy.ndarray>
>>> b.shape
()
>>> b
array(4.0)
>>> float(b)
4.0
```

```
>>> math.sqrt(b)
2.0
```

Note: an array with shape `(1,)` can also be converted to a float:

```
>>> c = np.array([3])
>>> c.shape
(1,)
>>> float(c)
3.0
```

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(3.)
>>> float(a)              # array with shpe ()
-0.13662231969702732   # converts to python float
>>> b = mexhat_np(np.arange(0, 11, 2))
>>> type(b)               # array with shape (6,)
<class numpy.ndarray>
>>> b
array([  1.53729366e+00,  -6.24150219e-01,
        -7.73556857e-03,  -8.19453296e-07,
        -1.22651811e-12,  -2.93540437e-20])
```

Scientific Python

SciPy (SCIentific PYthon)

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

```
stats        --- Statistical Functions
sparse       --- Sparse matrix
lib          --- Python wrappers to external
                 libraries
linalg       --- Linear algebra routines
signal       --- Signal Processing Tools
misc         --- Various utilities that don't
                 have another home.
interpolate --- Interpolation Tools
optimize     --- Optimization Tools
cluster      --- Vector Quantization / Kmeans
fftpack      --- Discrete Fourier Transform
io           --- Data input and output
integrate    --- Integration routines
lib.lapack   --- Wrappers to LAPACK library
special      --- Special Functions
lib.blas     --- Wrappers to BLAS library
```

Interpolation of data

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

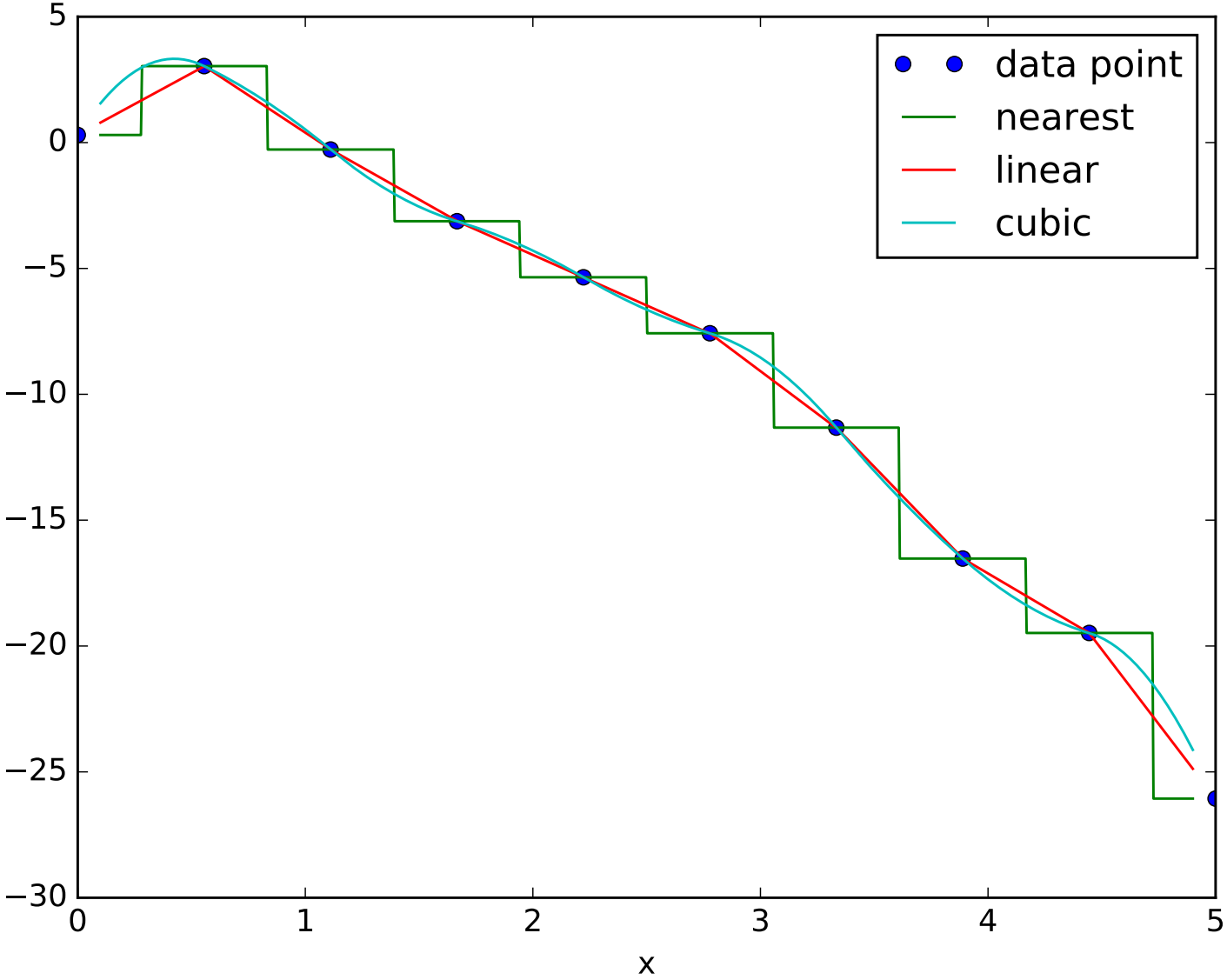
- $\rightarrow$ `y0 = scipy.interpolate.interp1d(x, y)` does this interpolation. Note that the function
- `interp1d` returns *a function* `y0` which will interpolate the x-y data for any given x when called as `y0(x)`.
- Data interpolation of $y_i = f(x_i)$ may be useful to
  - create smoother plots 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 $f(x)$
- Need to decide how to interpolate (nearest, linear, quadratic or cubic splines, ...)

Interpolation of data example

```
import numpy as np
import scipy.interpolate
import pylab

def create_data(n):
    """Given an integer n, returns n data points
    x and values y as a numpy.array."""
    xmax = 5.
    x = np.linspace(0, xmax, n)
    y = - x**2
    # make x-data somewhat irregular
    y += 1.5 * np.random.normal(size=len(x))
    return x, y
```

```
# main program
n = 10
x, y = create_data(n)

# use finer and regular mesh for plot
xfine = np.linspace(0.1, 4.9, n * 100)
# 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 piecewise constant func (p=2)
y2 = scipy.interpolate.interp1d(x, y, kind='quadratic')

pylab.plot(x, y, 'o', label='data point')
pylab.plot(xfine, y0(xfine), label='nearest')
pylab.plot(xfine, y1(xfine), label='linear')
pylab.plot(xfine, y2(xfine), label='cubic')
pylab.legend()
```

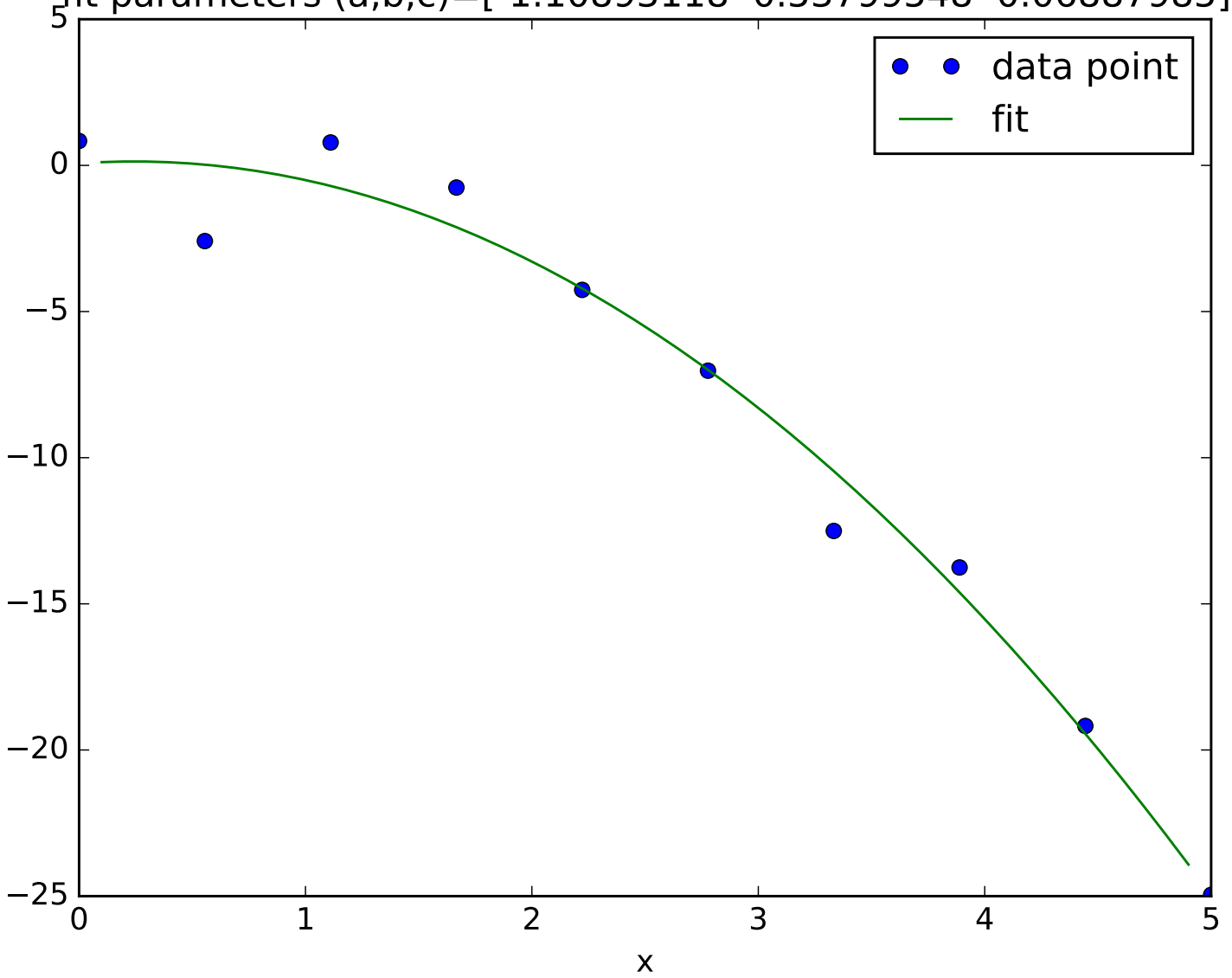
```
pylab.xlabel('x')
pylab.savefig('interpolate.pdf')
pylab.show()
```



Curve fitting example

```
import numpy as np
import scipy.optimize
import pylab

def create_data(n):
    """Given an integer n, returns n data points
    x and values y as a numpy.array."""
    xmax = 5.
    x = np.linspace(0, xmax, n)
    y = - x**2
    # make x-data somewhat irregular
    y += 1.5 * np.random.normal(size=len(x))
    return x, y

def model(x, a, b, c):  # Equation for fit
    return a * x ** 2 + b * x + c
```

```
# main program
n = 10
x, y = create_data(n)
# do curve fit
p, pcov = scipy.optimize.curve_fit(model, x, y)
a, b, c = p
# plot fit and data
xfine = np.linspace(0.1, 4.9, n * 5)
pylab.plot(x, y, 'o', label='data point')
pylab.plot(xfine, model(xfine, a, b, c), \
    label='fit')
pylab.title('fit parameters (a,b,c)=%s' % p)
pylab.legend()
pylab.xlabel('x')
pylab.savefig('curvefit2.pdf')
pylab.show()
```



Function integration example

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

which produces this output:

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

LAB10

Optimisation (Minimisation)

- Optimisation typically described as:
  given a function $f(x)$, find x_m so that $f(x_m)$ is the (local) minimum of f.
- To maximise $f(x)$, create a second function $g(x) = -f(x)$ and minimise $g(x)$.
- 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

Optimisation example

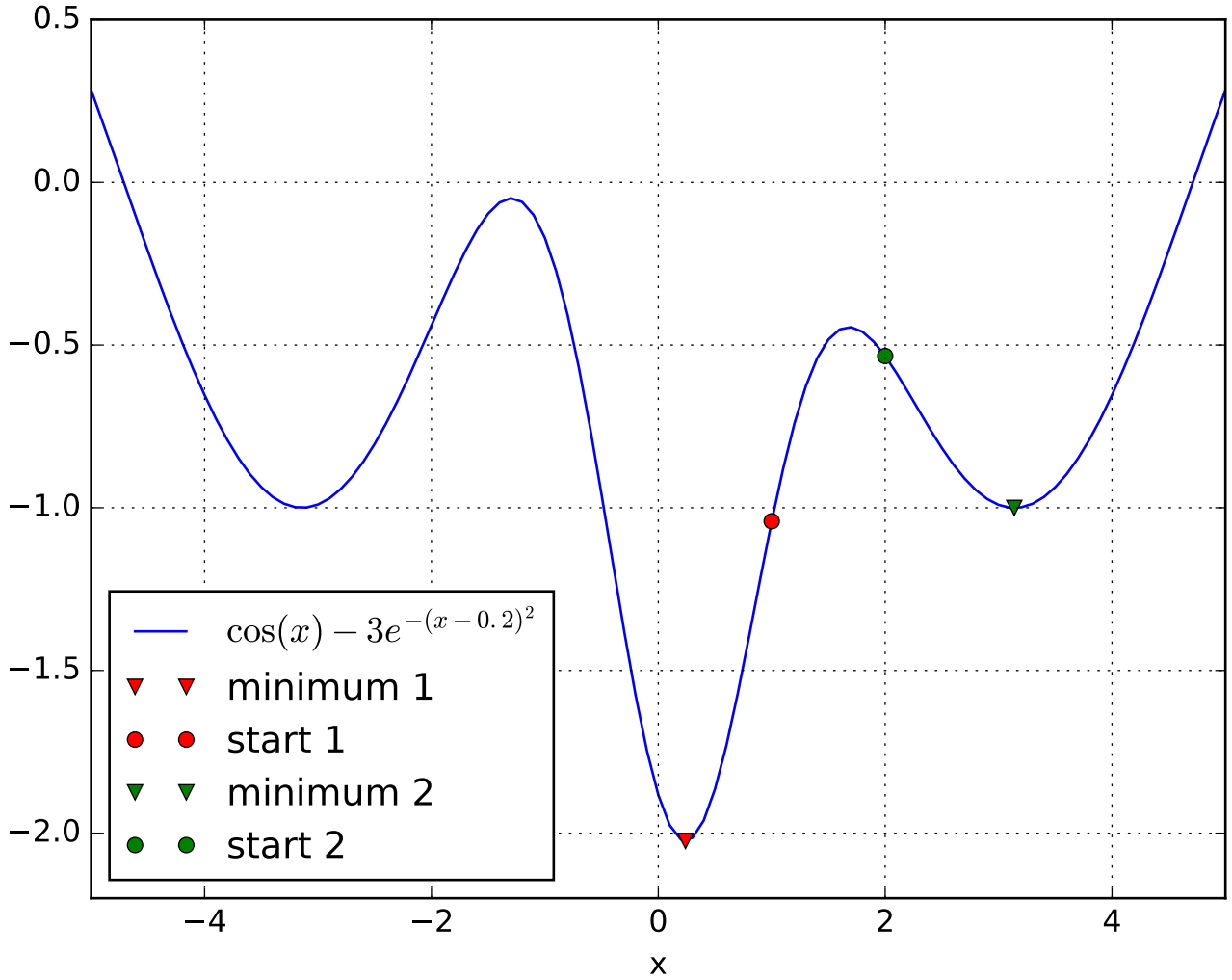
```
from scipy import arange, cos, exp
from scipy.optimize import fmin
import pylab

def f(x):
    return cos(x) - 3 * exp( -(x - 0.2) ** 2)

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

```
# plot function
x = arange(-10, 10, 0.1)
y = f(x)
pylab.plot(x, y, label='$\cos(x)-3e^{-(x-0.2)^2}$')
pylab.xlabel('x')
pylab.grid()
pylab.axis([-5, 5, -2.2, 0.5])

# add minimum1 to plot
pylab.plot(minimum1, f(minimum1), 'vr', label='minimum 1')
# add start1 to plot
pylab.plot(1.0, f(1.0), 'or', label='start 1')

# add minimum2 to plot
pylab.plot(minimum2,f(minimum2),'vg', label='minimum 2')
# add start2 to plot
pylab.plot(2.0,f(2.0),'og',label='start 2')
```

```
pylab.legend(loc='lower left')
pylab.savefig('fmin1.pdf')
pylab.show()
```

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]
```



LAB11

ODEs

Ordinary Differential Equations

- Many processes, in particular *time-dependent* processes, can be described as Ordinary Differential Equations (ODEs). This includes dynamics of engineering systems, quantum physics, chemical reactions, biological systems modelling, population dynamics, and many other models.
- ODEs have *exactly one* independent variable, and we assume for simplicity this is the time t.
- The easiest ODE type has one degree of freedom, y, which depends on the time t, i.e. $y = y(t)$. (For example temperature as a function of time, the distance a car has moved as function of time, the angular velocity of a rotating motor, etc.)

- In general, a vector $\mathbf{y}$ with k components can depend on the independent variable, in which case we are looking at a *system* of ordinary differential equations with k degrees of freedom.
- We are seeking the function $y(t)$ – this is *the solution* of the ODE.
- 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.

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y, t) \tag{14}$$

Interface `odeint`

- aim: solve
$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y, t)$$
- get access to “odeint”:

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

  ```
  ys = odeint(f, y0, ts)
  ```

  Input:
  - `f` is function `f(y, t)` that returns the right-hand side
  - `y0` is the initial value of the solution at time t_0
  - `ts` is a numpy `array` containing times t_i for which we would like to know the solution $y(t_i)$
    - the first value in the array has to be t_0 (with $y(t_0) = y_0$)

  Output:
  - `ys` is the numpy `array` that contains the solution

Using `odeint` – example 1

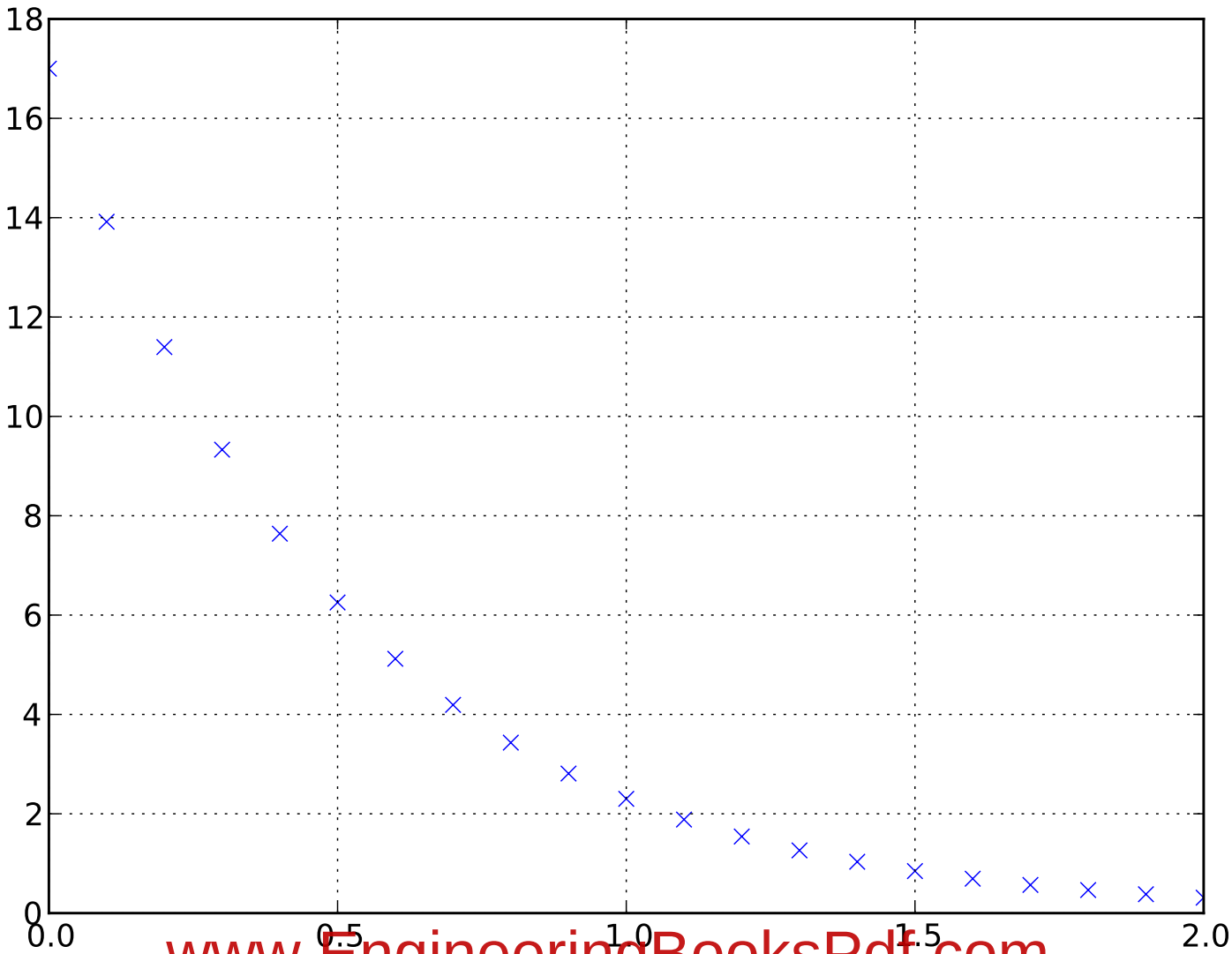
Require solution $y(t)$ from $t = 0$ to $t = 2$ of

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

```
import numpy as np
from scipy.integrate import odeint

def f(y,t):
    return -2 * y

ts = np.arange(0, 2.1, 0.1)
y0 = 17
ys = odeint(f, y0, ts)

import pylab
pylab.plot(ts, ys, 'x')
pylab.grid(); pylab.savefig('odeintexample1.pdf')
pylab.show()
```

Using `odeint` – example 1, solution

Solution:



Using `odeint` – example 2

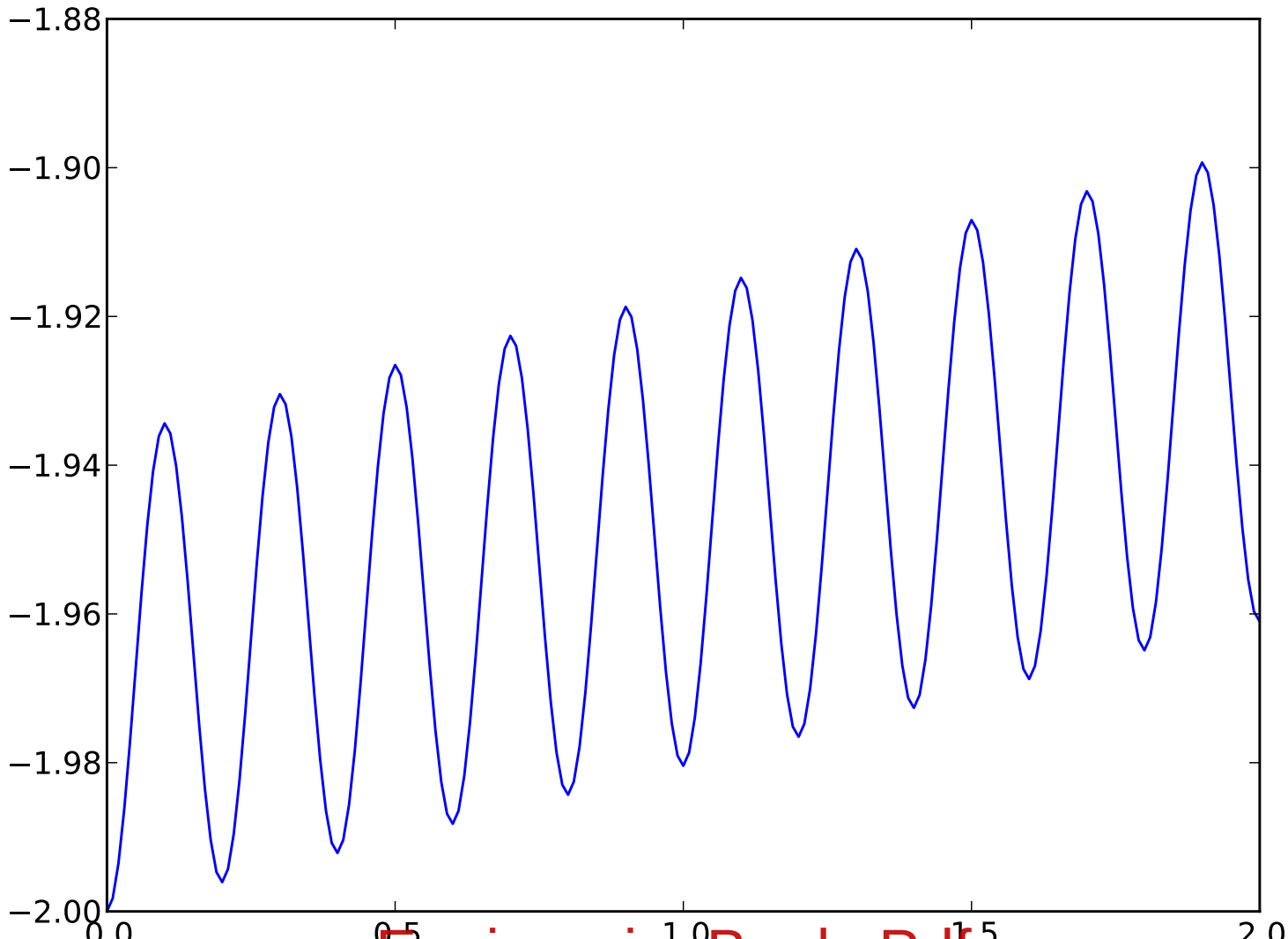
Require solution $y(t)$ from $t = 0$ to $t = 2$ of

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

```
import math
import numpy as np
from scipy.integrate import odeint

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

ts = np.arange(0, 2.01, 0.01)
y0 = -2
ys = odeint(f, y0, ts)

import pylab
pylab.plot(ts, ys)
pylab.savefig('odeintexample2.pdf'); pylab.show()
```

Using `odeint` – 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 `odeint`

- 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):

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

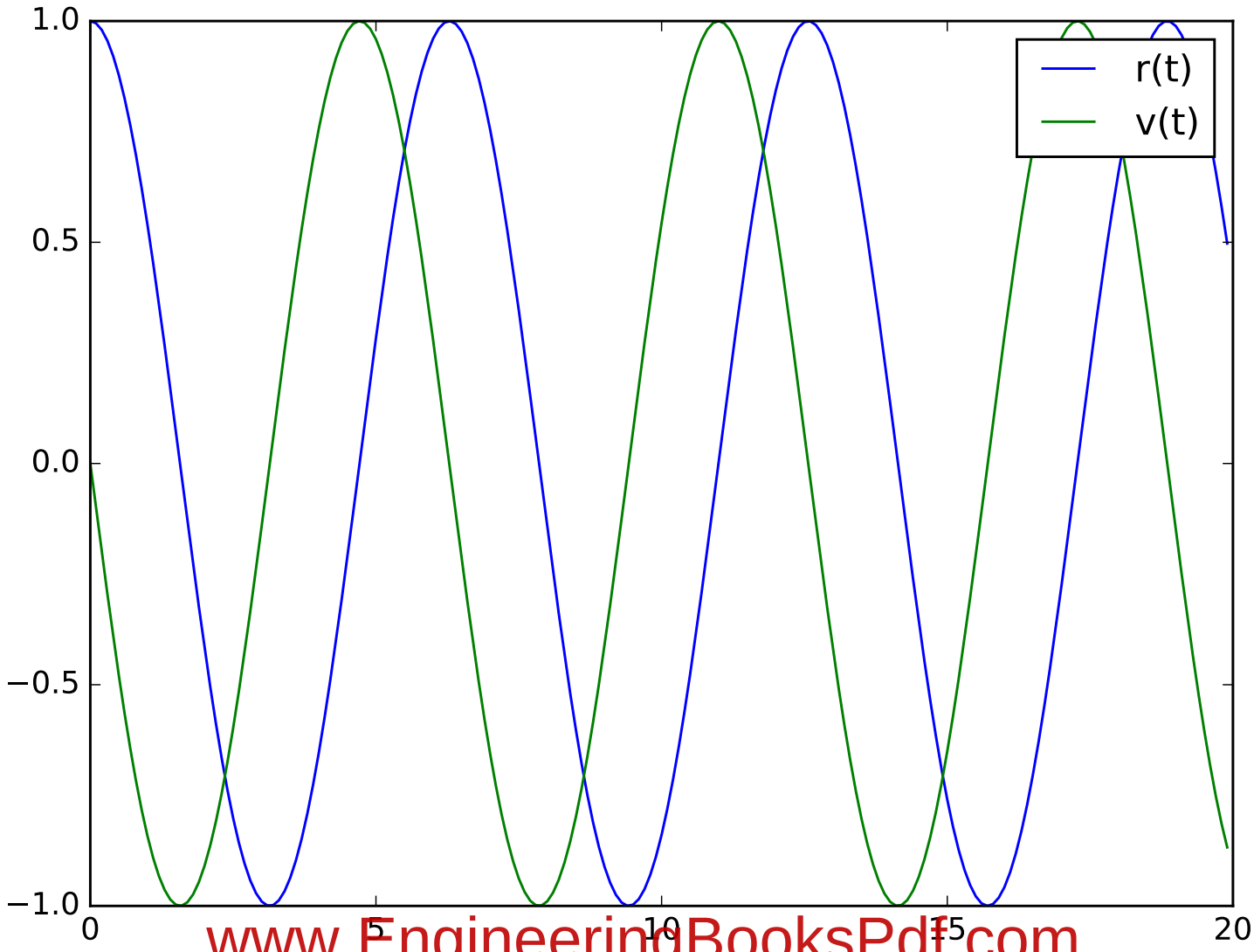
  - need to pack and unpack variables into the *state vector* $\mathbf{y}$:
  - Example harmonic oscillator:
    - decide to use this packing: $\mathbf{y} = (r, v)$
    - then $\mathbf{f}$ needs to return $\mathbf{f} = \left(\frac{\mathrm{d}r}{\mathrm{d}t}, \frac{\mathrm{d}v}{\mathrm{d}t}\right)$
  - `odeint` returns a vector $\mathbf{y}$ for every time step $\rightarrow$ a matrix
    - need to extract results for r and v from that matrix (rows are time, first column is r, second column is v) $\rightarrow$ see next slide

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

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

def f(y, t):                    # 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 = array([r0, v0])       # combine r and v into y

ys = odeint(f, y0, ts)     # solve ODEs

rs = ys[:, 0]              # extract results: r(t)
vs = ys[:, 1]              # extract results: v(t)
```

2nd order ODE – result

Solution (not annotated):



Summary 2nd order system

- Strategy:
  - transform one 2nd order ODE into 2 (coupled) first order ODEs
  - solve both first order ODEs simultaneously
- nothing conceptually complicated
- but need to use matrices (“arrays”) in Python to shuffle the data around.
- Warning: the meaning of y, x depends on context: often $x = t$ and $y = x$. It helps to write down equations before coding them.
- 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 $a = 0.7$
  - rabbit-fox-collision rate $c = 0.007$
  - fox death rate $b = 1$

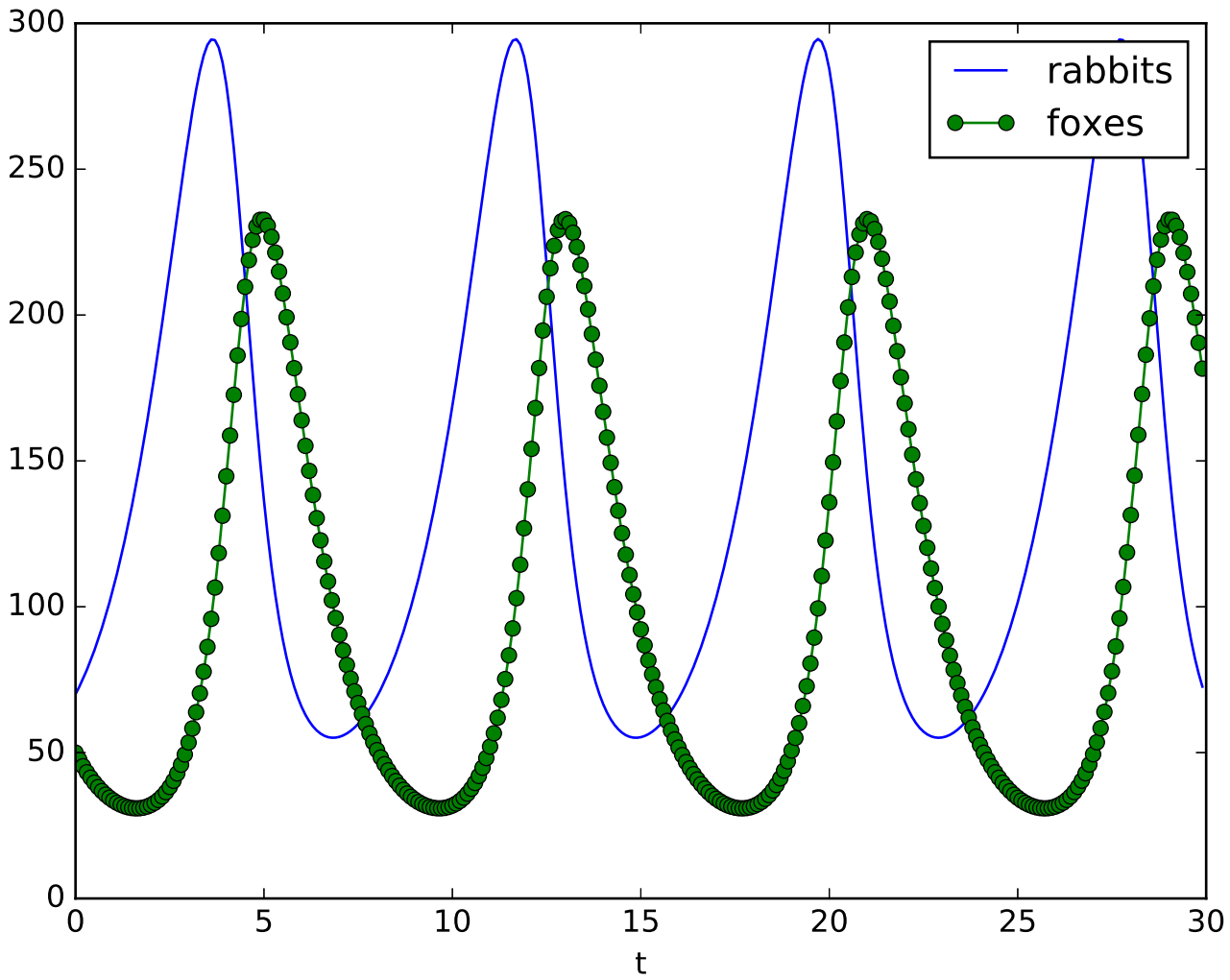
- Put all together in predator-prey ODEs

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

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

```
import numpy as np
from scipy.integrate import odeint

def rhs(y, t):
    a = 0.7;      c = 0.007;   b = 1
    p1 = y[0]
    p2 = y[1]
    dp1dt = a * p1 - c * p1 * p2
    dp2dt = c * p1 * p2 - b * p2
    return np.array([dp1dt, dp2dt])

p0 = np.array([70, 50])      # initial condition
ts = np.arange(0, 30, 0.1)

res = odeint( rhs, p0, ts ) # compute solution
```

```
p1 = res[:, 0]                  # extract p1 and
p2 = res[:, 1]                  # p2

import pylab                    # plot result
pylab.plot(ts, p1, label='rabbits')
pylab.plot(ts, p2, '-og', label='foxes')
pylab.legend()
pylab.xlabel('t')
pylab.savefig('predprey.eps')
pylab.savefig('predprey.png')
pylab.show()
```



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:

$$\mathbf{f} = \left(\frac{\mathrm{d}r_x}{\mathrm{d}t}, \frac{\mathrm{d}r_y}{\mathrm{d}t}, \frac{\mathrm{d}r_z}{\mathrm{d}t}, \frac{\mathrm{d}v_x}{\mathrm{d}t}, \frac{\mathrm{d}v_y}{\mathrm{d}t}, \frac{\mathrm{d}v_z}{\mathrm{d}t} \right) \tag{15}$$

Outlook examples

- Example: Molecular dynamics simulations have one set of 6 degrees of freedom as in equation (15) *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 (`odeint` and `ode`).

Sympy

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
3.14159265358979323846264338327950288419716939937
```

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
```

```
>>> 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
```

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
```

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
```

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"))
- In the Jupyter Notebook, run `sympy.init_printing()` to set up (LaTeX-style) rendered output

Testing

Testing - context

- Writing code is easy – debugging it is hard
- When debugging, we always *test* code
- 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:mixstrings.py

```
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:

    In []: mixstrings("Hello", "12345")
    Out[]: '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)

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

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

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

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

- tests are run if mixstrings.py runs on its own
    - No output if all tests pass (“*no news is good news*”)
- tests are not run if imported

Example 2: mixstrings-pytest

```
import pytest

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:

    In []: mixstrings("Hello", "12345")
    Out[]: '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)

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

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

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

if __name__ == "__main__":                    # need filename
    pytest.main("-v mixstrings-pytest.py")    # to test here
```

- pytest finds functions starting with `test_` automatically
- and executes them. Output when tests pass:

```
$> python mixstrings-pytest.py
======================== test session starts ========================
platform darwin -- Python 3.5.2, pytest-2.9.2
collected 3 items
mixstrings-pytest.py::test_mixstrings_basics PASSED
mixstrings-pytest.py::test_mixstrings_empty PASSED
mixstrings-pytest.py::test_mixstrings_different_length PASSED
===================== 3 passed in 0.01 seconds ======================
```

- `pytest` provides beautiful error messages when tests fail. If you can use `pytest`, do it.

py.test

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 3.5.2, pytest-2.9.2
collected 3 items

mixstrings.py::test_mixstrings_basics PASSED
mixstrings.py::test_mixstrings_empty PASSED
mixstrings.py::test_mixstrings_different_length PASSED
===================== 3 passed in 0.01 seconds ======================
```

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

Advanced Example 3: factorial.py

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

```
import math
import pytest

def factorial(n):
    """ Compute and return n! recursively.
    Raise a ValueError if n is negative or non-integral.

    >>> from myfactorial import factorial
    >>> [factorial(n) for n in range(5)]
    [1, 1, 2, 6, 24]
    """

    if n < 0:
        raise ValueError("n should be not-negative, but n = {}"
```

```
                        .format(n))

    if isinstance(n, int):
        pass
    else:
        raise ValueError("n must be integer but type(n)={}"
                         .format(type(n)))

    # actual calculation
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

def test_basics():
    assert factorial(0) == 1
    assert factorial(1) == 1
    assert factorial(3) == 6

def test_against_standard_lib():
    for i in range(20):
```

```
        assert math.factorial(i) == factorial(i)

def test_negative_number_raises_error():
    with pytest.raises(ValueError):     # this will pass if
        factorial(-1)                   # factorial(-1) raises
                                        # an ValueError

def test_noninteger_number_raises_error():
    with pytest.raises(ValueError):
        factorial(0.5)
```

Output from successful testing:

```
$> py.test -v factorial.py
======================== test session starts ========================
platform darwin -- Python 3.5.2, pytest-2.9.2
collected 4 items

factorial.py::test_basics PASSED
factorial.py::test_against_standard_lib PASSED
factorial.py::test_negative_number_raises_error PASSED
```

```
factorial.py::test_noninteger_number_raises_error PASSED
==================== 4 passed in 0.01 seconds ====================
```

Notes on pytest

- Normally, we call an executable **`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:

```
$> py.test -v factorial.py mixstrings.py
======================== test session starts ========================
platform darwin -- Python 3.5.2, pytest-2.9.2 -- python
collected 7 items
factorial.py::test_basics PASSED
factorial.py::test_against_standard_lib PASSED
factorial.py::test_negative_number_raises_error PASSED
factorial.py::test_noninteger_number_raises_error PASSED
mixstrings.py::test_mixstrings_basics PASSED
mixstrings.py::test_mixstrings_empty PASSED
mixstrings.py::test_mixstrings_different_length PASSED
===================== 7 passed in 0.01 seconds ======================
```

Testing summary

- Unit testing, integration testing, regression testing, system testing
- absolute key role in modern software engineering
- good practice to always write tests when code is written
- bigger projects have ”continuous integration testing”
- ”eXtreme Programming” (XP) philosophy suggests to write tests before you write code (”test-driven-development (TDD)”)
- More on this in FEEG6002 Advanced Computational Methods

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

Object Oriented 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("{:2}:{:2}".format(self.hour, self.min))

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

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

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

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

    print("The time is %d hours and %d minutes." % (t.hour, t.min))
```

which 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 created
- all methods in a class need `self` as the first argument. Self represents the object. (This will make more sense later.)
- 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 → property).

Data Hiding

- A well designed class provides methods to get and set attributes.
- These methods define the *interface* to that class.
- This allows
  - to perform error checking when values are set, and
  - to hide the implementation of the class from the user. This is good because
    - the user doesn't need to know what is going on behind the scenes
    - we can change the implementation of the class without changing the interface.
- The next slides show an extended version of the `Time` class with such get and set methods.
- 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 recommended over set and get methods.

Example 2: a class to deal with time

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

    def setHour(self, hour):
        if 0 <= hour <= 23:
            self.hour = hour
        else:
            raise ValueError("Invalid hour value: %d" % hour)

    def setMin(self, min):
        if 0 <= min <= 59:
            self.min = min
        else:
            raise ValueError("Invalid min value: %d" % min)
```

```
    def getHour(self):
        return self.hour

    def getMin(self):
        return self.min


    def print24h(self):
        print("{:2}:{:2}".format(self.getHour(),
                                 self.getMin()))
    def print12h(self):
        if self.getHour() < 12:
            ampm = "am"
        else:
            ampm = "pm"

        print("{:2}:{:2} {}".format(self.getHour() % 12,
                                    self.getMin(), ampm))


if __name__ == "__main__":
```

```
    t = Time(15, 45)

    print("print as 24h: "),
    t.print24h()
    print("print as 12h: "),
    t.print12h()
    print("that is %d hours and %d minutes" % \
          (t.getHour(), t.getMin()))
```

which produces

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

Data Hiding (2)

- providing *set* and *get* methods for attributes of an object
  - prevents incorrect data to be entered
  - ensures that the internal state of the object is consistent
  - hides the implementation from the user (more black box),
  - and make future change of implementation easier
- there are more sophisticated ways of “hiding” variables from users: using Python *properties* we can bind certain functions to be called when attributes in the class are accessed. (See for example here).

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.

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

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

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

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

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

    print("String representation of the object t1: %s" % t1)
    print("Representation of object = %r" % t1)

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

Output:

```
String representation of the object 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 define a *base class* and *derive* other classes from it.
- This is called *inheritance*. The next slides show examples

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 "[ {:2}:{:2} ]".format(self.hour, self.min)

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

```
            return True
        else:
            return False

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 "[{:2}:{:2}{}]".format(self.hour % 12,
                                      self.min, ampm)


if __name__ == "__main__":
```

```
    t3 = TimeUK(15, 45)
    print("TimeUK object = %s" % t3)
    t4 = Time(16, 15)
    print("Time   object = %s" % t4)
    print("compare t3 and t4: ")
    if t3 > t4:
        print("t3 is greater than t4")
    else:
        print("t3 is not greater than t4")
```

Output:

```
TimeUK object = [ 3:45pm]
Time   object = [ 16:15 ]
compare t3 and t4:
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

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] != "__"])
```

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).

Objects in Python

- *All* "things" in Python are objects, including numbers, strings and functions.
- Try this at the prompt:

```
>>> 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.
- There is a number of helpful concepts, including data hiding and operator overloading.
- Classes can provide get and set methods and hide their implementation.
- Classes can be derived from other classes.

Some software engineering observations

- OOP needs some time to get used to.
- Good use of OOP
  - makes large codes easier to maintain,
  - encourages re-use of existing code,
  - requires some thought (finding the right classes for a given problem can be difficult).

Note: Fortran 2003 supports Object Oriented Programming.

Some programming languages

Some Programming Languages for Computational Science

see also: `http://www.levenez.com/lang/`

for an overview of general programming languages

Requirements

In the early days ...

- computers were slow and rare
- computing time was very precious
- invest effort to run programs as fast as possible

Nowadays ...

- increasingly more computing power available
- major cost factor is the time it takes to write (and debug and test) software

This is not always true (because it depends on the application) but it seems to be generally correct.

Fortran

- FORmula TRANslator (design goal is to translate formulae)
- 1957: Fortran I
- commonly used:
  - Fortran 66 and 77
  - Fortran 90 and 95
    (“matrix notation”)
- compiled language, “low level”
- inbuilt complex numbers $\rightarrow$ popular with scientists
- very fast
- many numerical libraries are written in Fortran
- Fortran 2003 introduced “Objects”

C

- developed 1972 in Bell laboratories
- 1978 Kerninghan & Ritchie C (often called K&R)
- 1989 ANSI C
- design goals:
  - economy of expression
  - absence of restriction
- compiled language, “low level”
- UNIX/Linux written in C
- no in-built complex numbers (pre C99)
- very fast
- some numerical libraries are written in C
- general purpose language (in comparision to Fortran)

- developed 1983 Bjarne Stroustrup
- compiled language, “low level”
- Object Oriented
- set of higher-level tools (STL)
- fast
- C is subset of C++ but
- knowing C does not mean knowing C++
- general purpose language

MATLAB

- MATrix LABoratory (1984)
- started as collection of linear algebra functions
- scripting language grew to combine these functions
- visualisation features were added
- this is what we call MATLAB today
- the MATLAB scripting language is interpreted and slow
- the numerical libraries that come with MATLAB are compiled and fast
- designed for numerical work (and very good at this)
- can be fast if used carefully
- “high-level language” $\rightarrow$ higher coding efficiency
- commercial product
  - need to pay
  - users can not see (and check) source code
- very popular in engineering community

Python

- 1990 Python (named after Monty Python)
- high coding efficiency due to
  - “high-level language”
  - interpreted
  - simple syntax and clean design
  - huge tool box, including GUIs, internet, XML, data bases, graphics, gaming, numerical processing
- general purpose language
- easy to include compiled Fortran and C code
  - re-use of existing libraries
  - way to make Python programs fast
- fully supports Object Orientation
- performance comparable to MATLAB
- growing popularity in commerce, science and industry

Comparison

Selected criteria:

	Fortran	C	C++	Matlab	Python
performance	+	+	+	○	○
object orientation	–	–	+	–	+
exceptions	–	–	+	–	+
open source	+	+	+	–	+
easy to learn	○+	○	○–	+	+

legend:

+	=	good/yes
○	=	so-so
–	=	poor/no

How do Python and MATLAB relate?

- both MATLAB and Python share (good) ideas:
  - provide high-level scripting language (slow) to glue together
  - fast compiled libraries
- Some differences are:
  - MATLAB is written for Engineering and Computational Science tasks and leading in combining computation and visualisation.
  - MATLAB is a commercial product.
  - Python is a modern, general purposes language and easier to extend.
  - “Large” projects are better advised to use Python as the glueing language.

Summary

- Need Fortran and C for extreme speed
- Need high-level language to reduce development time (that is time for writing the program)
  - This is particularly important in a research environment where requirements often change.
- Strategy:
  - write parts of the code that take most of the execution time in C/Fortran
  - write the remaining program (for visualisation, data input output, ...) in high-level language.
  - For large computational programs, generally $>$ 99% of the CPU time are spend in a few % of the code.

What language to learn next?

What language to learn next?

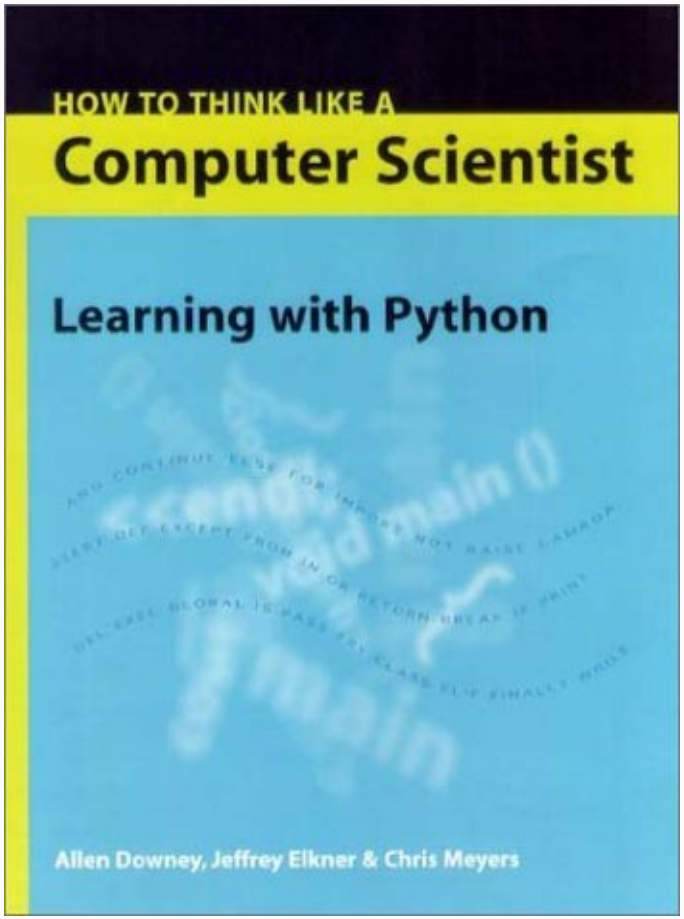
- ...it all depends what you want to achieve:
- To learn C or Fortran, get a book or on-line tutorial.
- To learn object oriented programming (OOP), read “How to think like a computer Scientist” (for Python), or try “Python – how to program” (see next slides).
- To learn C++, learn OOP using Python first, then switch to C++.
- To learn Java or C#, you should consider learning OOP using Python first (but you could risk switching language straight away).

Note:

- Python provides an excellent platform for all possible tasks

$\Rightarrow$ it could well be all you need for some time to come.

Further reading



- "How to Think Like a Computer Scientist: Learning with Python". (ISBN 0971677506) free at http://greenteapress.com/wp/think-python-2e/
- Very systematic presentation of all important programming concepts, including OOP.
- Aimed at first year computer science students.
- Recommended if you like programming and want to know more about the concepts.

Other tools and topics

Tools to extend your computational toolkit / suggested self-study topics

- Systematic testing (py.test or nose) and
- Test Driven Development (TDD)
- Version control (try Mercurial or Git)
- Automate everything
- IPython Notebook
- LaTeX (professional document creation)
- Cool editor (Emacs? Sublime? ...)

Acknowledgements

Thanks go to Ondrej Hovorka and Neil O’Brien for contributing to these slides.