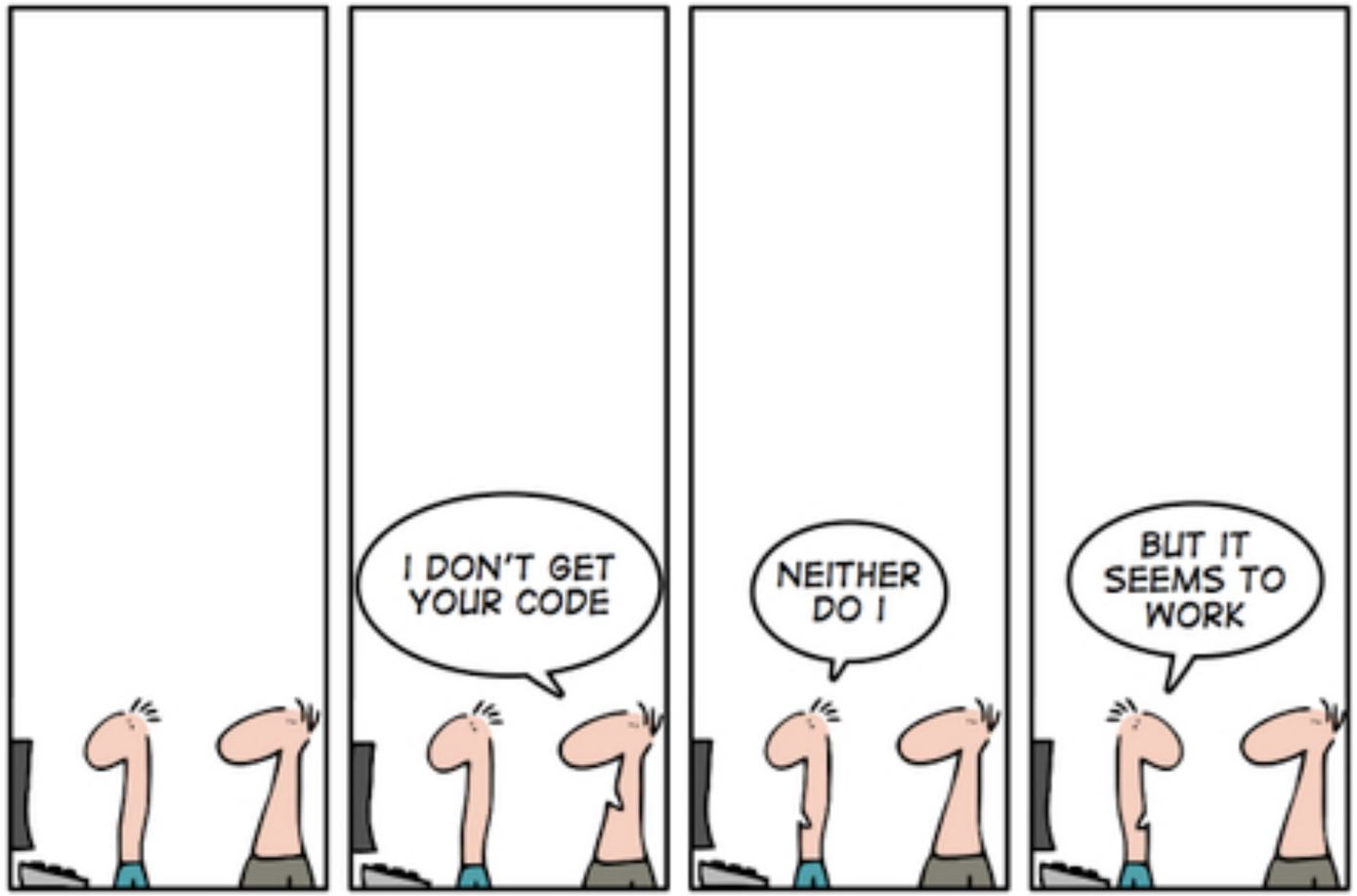
Leibniz Universität Hannover

IQ Institute of Quantum Optics



THE ART OF PROGRAMING

[http://geekandpoke.com]

Science is knowledge which we understand so well that we can teach it to a computer; and if we don't fully understand something, it is an art to deal with it.

Donald E. Knuth, *Computer Programming as an Art*

PhoenixD

Leibniz Universität Hannover

IQ Institute of Quantum Optics

Introduction to Programming in Python

— how to (not) do things + exercises —

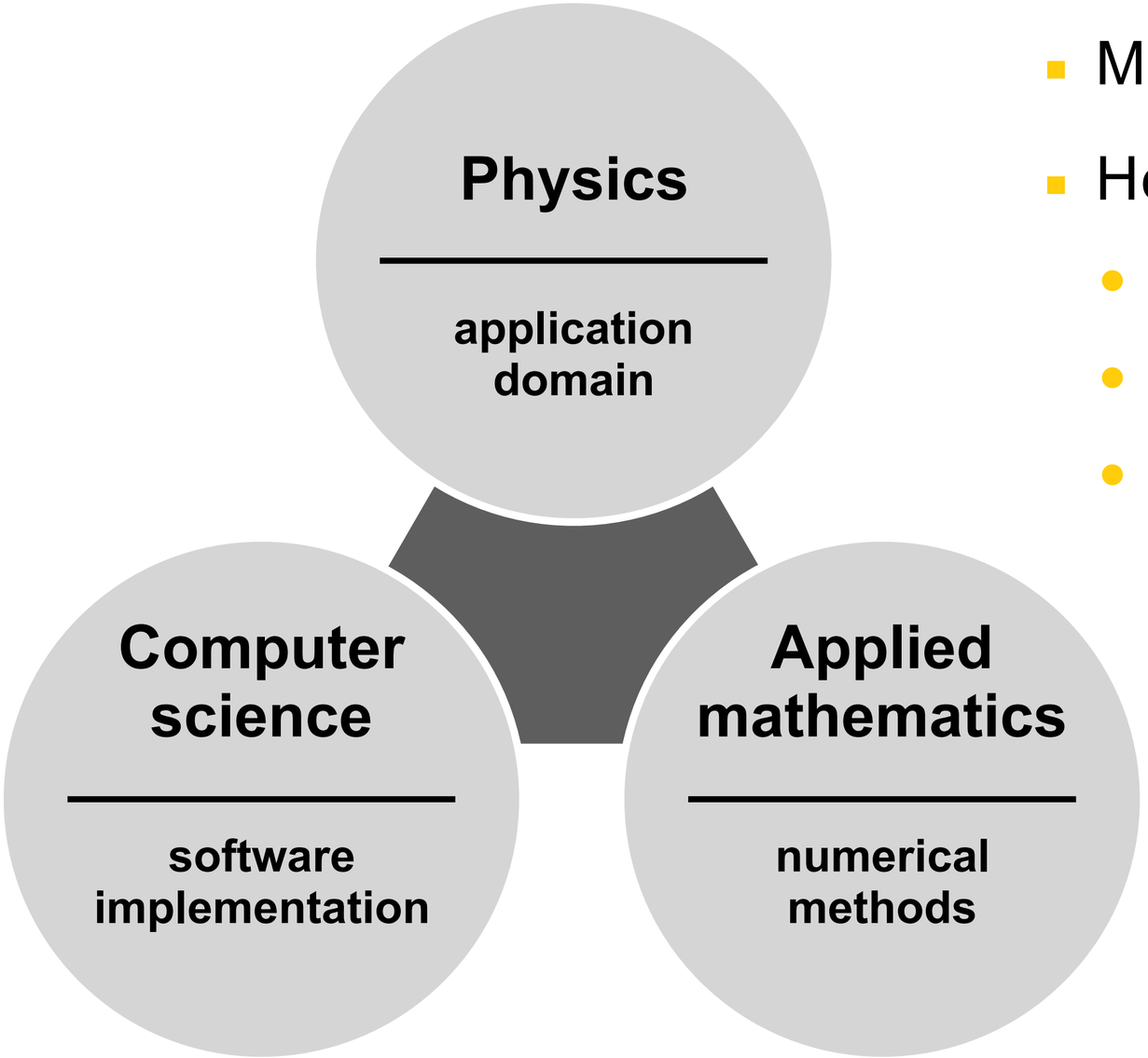
O. Melchert
(assistence: S. Willms)

PhoenixD

Outline

- Preface
- Part I — programming methods
- Part II — implementation variants
- Part III — programming paradigms

Computational Physics



- Multidisciplinary field of science
- Here:
  - Simplified exemplary applications
  - Numerical methods given upfront
  - Focus on *how to do things* using

python™

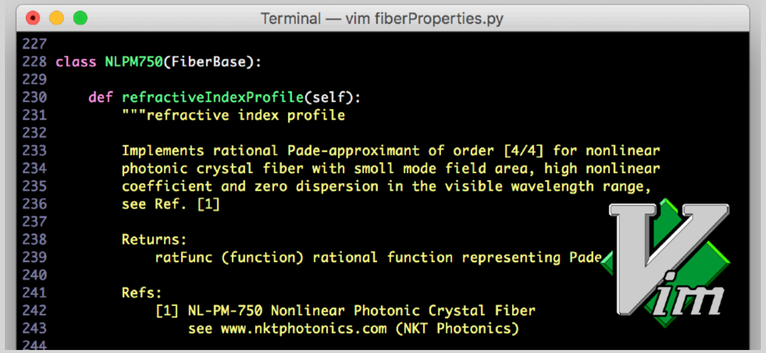
How to write code at all?

Solutions to the development logistics problem:

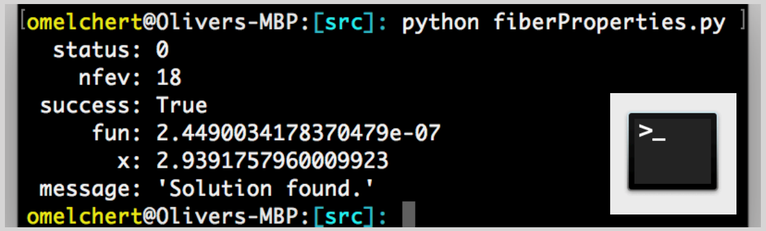
(1) use a *terminal* and a *text editor*

pro: super flexible

con: requires some training



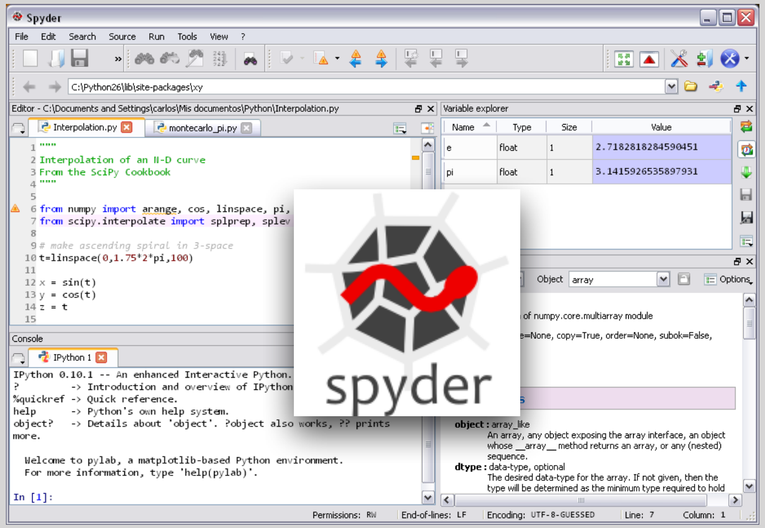
[http://www.vim.org]



(2) use an *IDE*

pro: easy to use

con: less flexible all-in-one solution

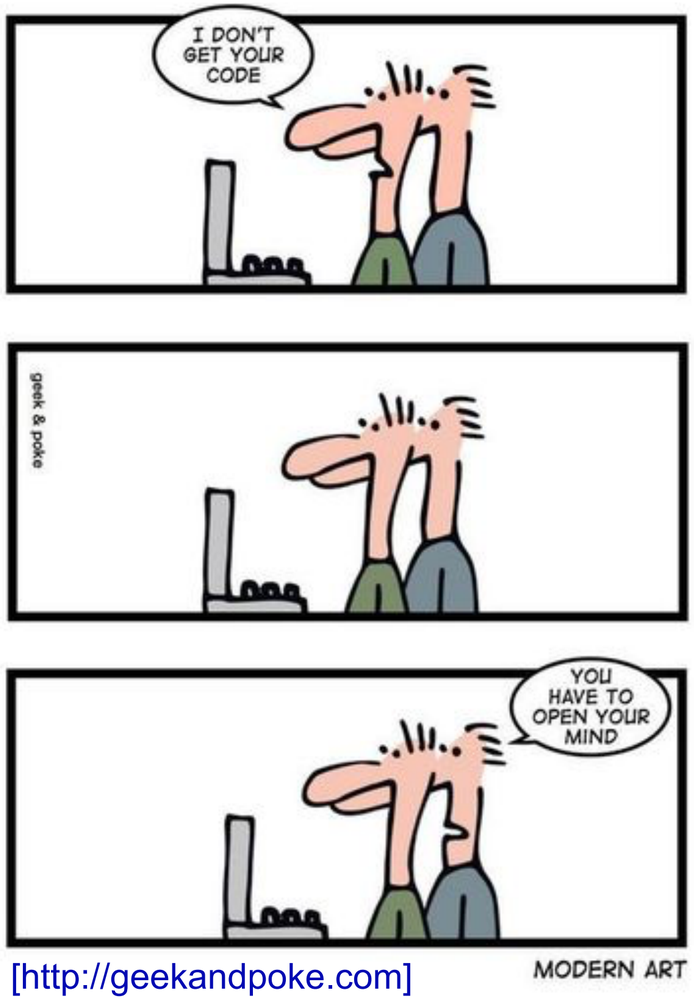


[https://pythonhosted.org/spyder/]

If you have no experience or preference, see what your peers and supervisors use!

How to write python code?

```
Python 2.7.10 (default, Feb  7 2017, 00:08:15)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.34)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```



[http://geekandpoke.com]

Core guiding principle: others should be able to understand what you did and why!

[P. Goodliffe, Becoming a Better Programmer, O'Reilly (2014)]

Brief general introduction: [W. S. Noble, A Quick Guide to Organizing Computational Biology Projects, PLoS Comp. Bio. (2009)]

Basics

- Data structures
  - *Build-in* data types
    - Lists
    - Sets
    - Dictionaries
  - *Custom* data types
    - Classes
- Functions and modules
- Libraries we will use:
  - numpy
  - matplotlib

Part I — Programming Methods

Recursive vs. dynamic programming

- Occasionally problems are formulated as recurrence relations
- Example: Construction of ***Fibonacci sequence***

Recurrence relation: $F_n = F_{n-1} + F_{n-2}$

Initial condition: $F_0 = 0$

$F_1 = 1$

First few numbers in the sequence:

$$n: \ 0, \ 1, \ 2, \ 3, \ 4, \ 5, \ 6, \ 7, \ 8, \ 9, \ldots$$

$$F_n: \ 0, \ 1, \ 1, \ 2, \ 3, \ 5, \ 8, \ 13, \ 21, \ 34, \ldots$$

Recursive vs. dynamic programming

- Recursive *top-down* solution:

```
1 def fib_recursive(n):
2     return fib_recursive(n-1) + fib_recursive(n-2) if n>=2 else n
```

  - Subdivide large problem into smaller problems
  - Solves smaller problems until complete

→ Typically slow!

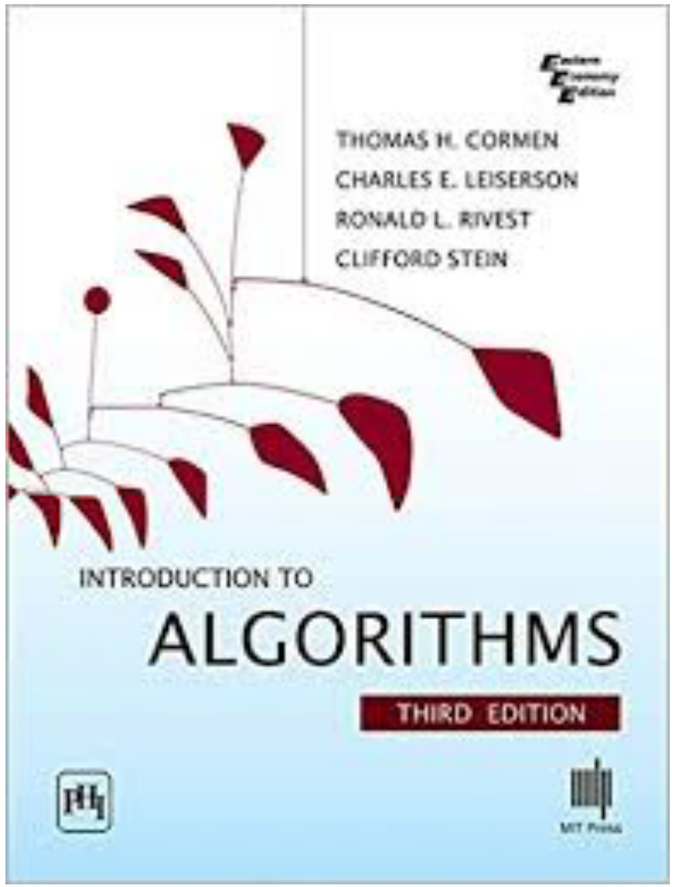
- Dynamic *bottom-up* solution:

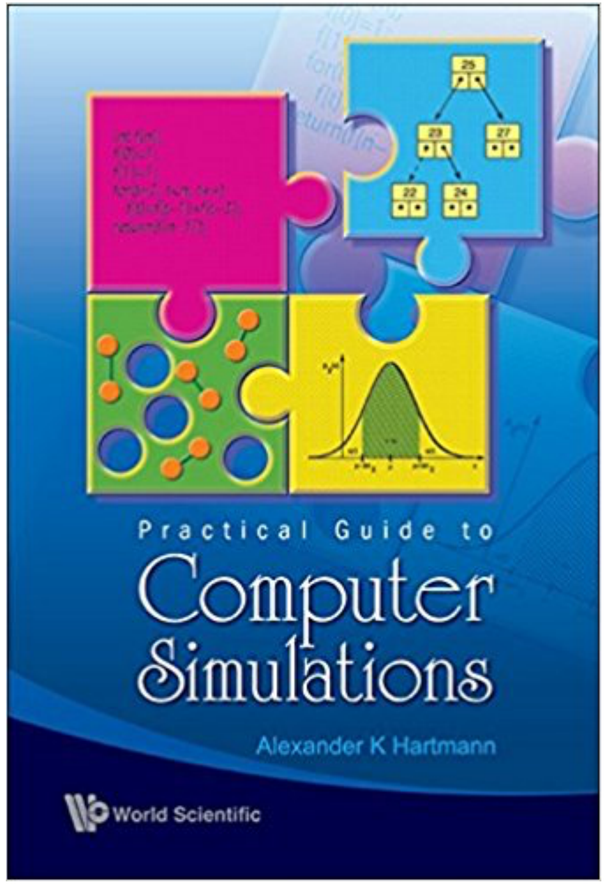
```
1 def fib_dynamic(n):
2     fSeq = [0]*(n+1)
3
4     if n >=1:
5        fSeq[1] = 1
6        for i in range(2,n+1):
7           fSeq[i]=fSeq[i-1]+fSeq[i-2]
8
9     return fSeq[-1]
```

  - Solve small subproblems
  - Combine intermediate solutions to yield result

→ Typically fast!

Further reading





Recursive vs. dynamic programming

P1 — The dynamic programming variant is not the only non-recursive method to solve a recurrence relation. Consider the example of the Fibonacci sequence. You might have noticed that the dynamic programming algorithm could be mimicked without using an array at all.

(A) Devise a simple *iterative* algorithm that solves the Fibonacci recurrence relation by completing the function **`fib_iterative(),`** pre-defined in the module file **`fibSeq_dynamic.py`**. Use a small set of variables only! Before you start coding, figure out how many variables you need to maintain as you go along!

(B) How does the resulting iterative algorithm perform? What do you expect? Amend the script **`main_timing.py`** to find out!

Part II — Implementation Variants

Sequential vs. vectorised programming

- Solution of partial differential equations (PDEs) requires iterative update of field-values at mesh points … this can be done incredibly inefficient
- Example: **1D diffusion equation**

Model

PDE: $$\partial_t u(x,t) - D\ \partial_x^2 u(x,t) = 0$$

Initial condition: $$u(x, t=0) = u_0(x)$$

Boundary conditions: $$u(x = x_{\min}, t) = 0$$
$$u(x = x_{\max}, t) = 0$$

- Wide range of applications in physics, biology and financial science

Sequential vs. vectorised programming

- Simple (*explicit*) finite difference approximation:
  - Discretise space and time coordinates:

$$x \longrightarrow \{x_i\}_{i=0}^{N_x}; \quad x_i = x_{\min} + i\Delta x;\ \Delta x = \frac{x_{\max} - x_{\min}}{N_x + 1}$$

$$t \longrightarrow \{t_n\}_{n=0}^{N_t}; \quad t_i = n\Delta t;\ \Delta t = \frac{t_{\max}}{N_t + 1}$$

  - Require PDE to be fulfilled at mesh-points, where $u_i^n \equiv u(x_i, t_n)$
  - Forward time, centered space discretisation scheme to yields

$$u_i^{n+1} = u_i^n + D\frac{\Delta t}{\Delta x^2}\left[u_{i-1}^n - 2u_i^n + u_{i+1}^n\right]$$

for interior points $i = 1 \ldots N_x - 1$

$$u_0^{n+1} = u_{N_x}^{n+1} = 0$$

for boundary values

Method

Sequential vs. vectorised programming

- Sequential implementation:

Here, circularity of lists can lead to bugs that are difficult to find!

Generally slow!

Mapping 1

```
def FTCS_sequential(x, t, u0, D=0.5):
    dt = t[1]-t[0]; dx = x[1]-x[0]
    C = D*dt/dx/dx
    um = np.copy(u0)
    u = np.copy(u0)

    for ti in range(t.size):
        for i in range(1,x.size-1):
            u[i] = um[i] + C*(um[i-1] - 2*um[i] + um[i+1])
        u[0]=0; u[-1]=0
        um[:]=u

    return u
```

- Vectorised implementation:

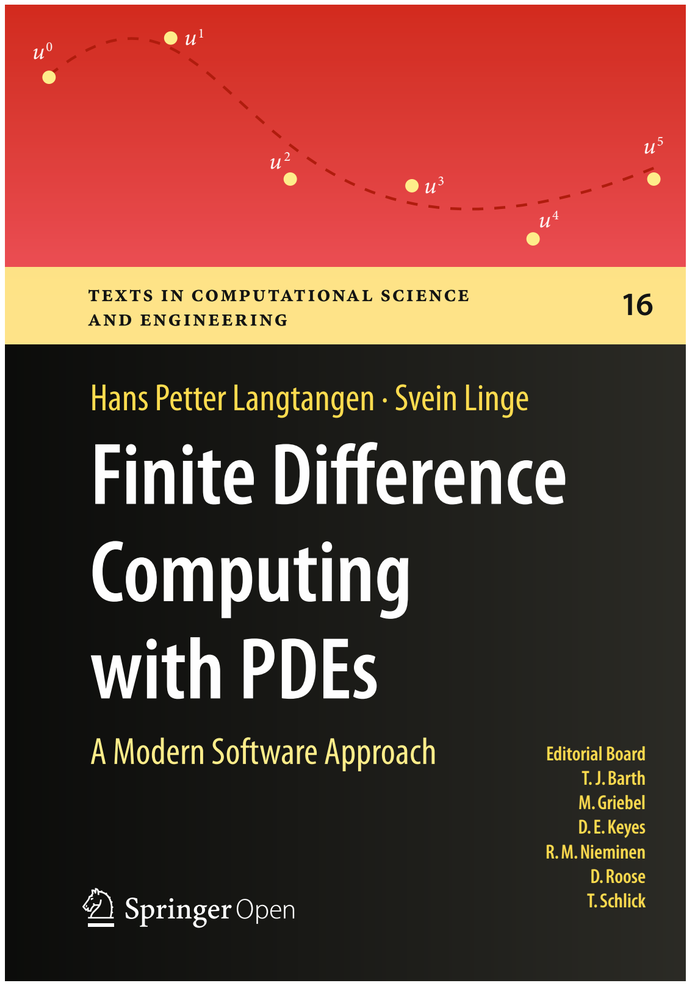
Uses start:stop - slicing and works on displaced arrays!

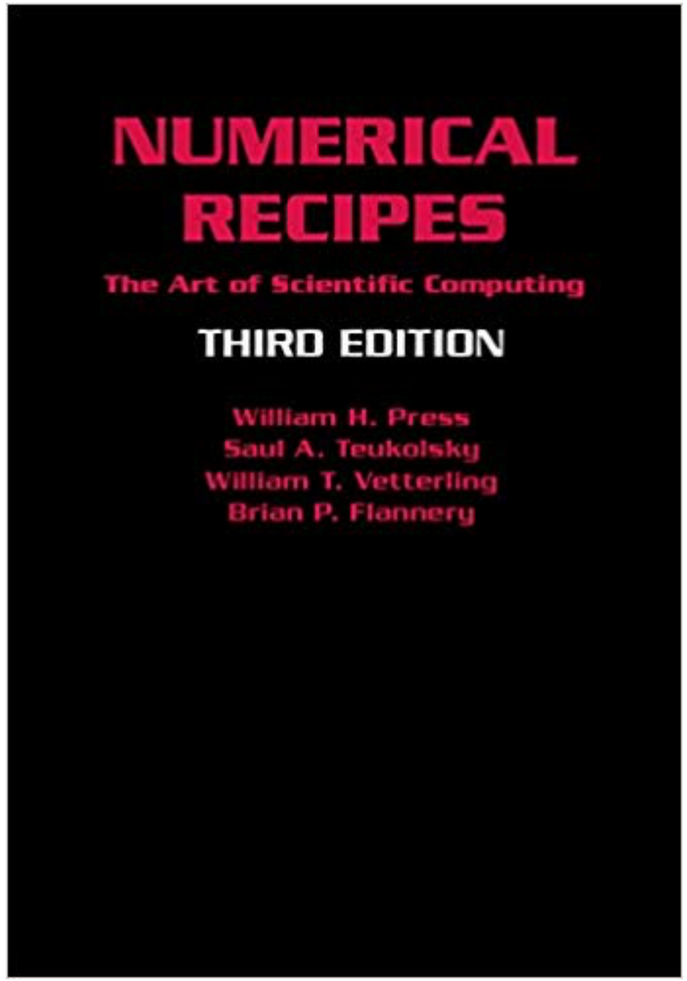
Generally fast!

Mapping 2

```
def FTCS_vectorized(x, t, u0, D=0.5):
    dt = t[1]-t[0]; dx = x[1]-x[0]
    C = D*dt/dx/dx
    u = np.copy(u0)

    for ti in range(t.size):
        u[1:-1] = u[1:-1] + C*(u[:-2] - 2*u[1:-1] + u[2:])

    return u
```

Further reading





Sequential vs. vectorised programming

P2 — Consider the script **`main_1DDiffusionEq.py`** that implements a simple example employing the finite-difference schemes presented in the lecture. *Wade* through the script!

(A) An important build-in data structure is the *dictionary*. In function **`fetchSolver()`** such a dictionary is used to emulate the popular switch-command provided by other languages. What does the switch accomplish?

(B) How does the implementation **`FTCS_vectorized()`** impose the boundary conditions?

(C) Extend the functionality of the provided code by implementing a function called **`setInitialCondition()`**, that returns the initial condition instead of hard-coding it (as in the provided implementation).

Sequential vs. vectorised programming

(C) Perform a v*erification test:* amend the provided code so that the 1D analytic result

$$u(x,t) = \frac{1}{\sqrt{4\pi Dt}} \exp(-x^2/4Dt)$$

is shown alongside the numerical solution. Do you find a good fit?

(D) Assess the *stability* of the FTCS scheme. The implemented scheme is stable as long as

$$C = \frac{D\Delta t}{\Delta x^2} \leq \frac{1}{2}$$

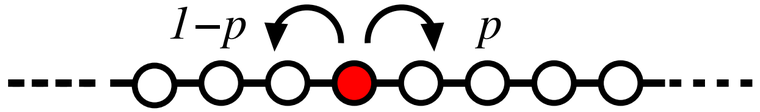
vary the time-increment so as to probe the limit of the above stability criterion. What do you *expect* (consult the model; use pen and paper)? What do you *observe*?

Part III — Programming Paradigms

Object oriented programming

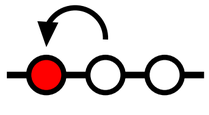
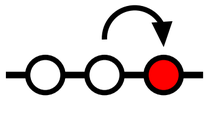
- Some problems naturally asks for an object-oriented (OO) implementation
- Paradigmatic example: **1D random walk**

Random experiment:



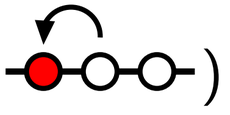
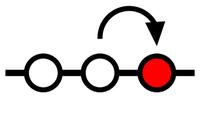
(outcome not predictable)

Sample space:

$\Omega = \{$  ,  $\}$

(set of elementary events)

Random variable:

$\Delta X($  $) = -1$ $\Delta X($  $) = 1$

(function assigning a numerical value to each elementary event)

- Here: generate 1D random walks using Monte Carlo simulation approach

Object oriented programming

- Simple 1D random walk class:

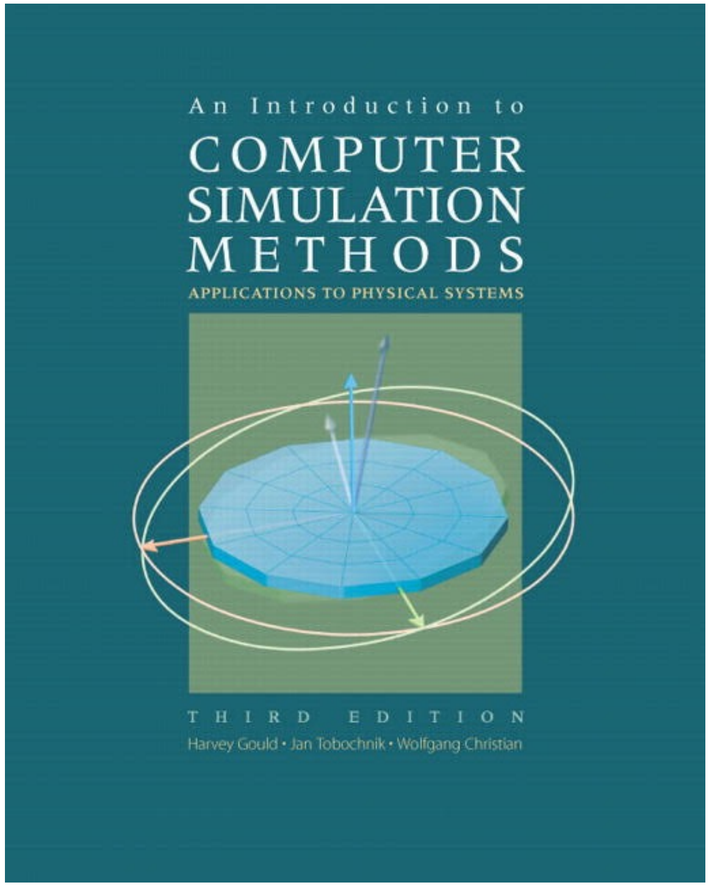
```
from random import random, seed

class RandomWalk(object):
    def __init__(self, x0, p=0.5):
        self.n = 0
        self.x0 = x0
        self.x = x0
        self.dx = lambda: -1 if random()<p else 1

    def step(self):
        self.x += self.dx()
        self.n += 1

def main():
    N = 20
    x0 = 0
    p = 0.5

    W = RandomWalk(x0,p=0.5)
    for n in range(N):
        W.step()
        print(n,W.x)

main()
```

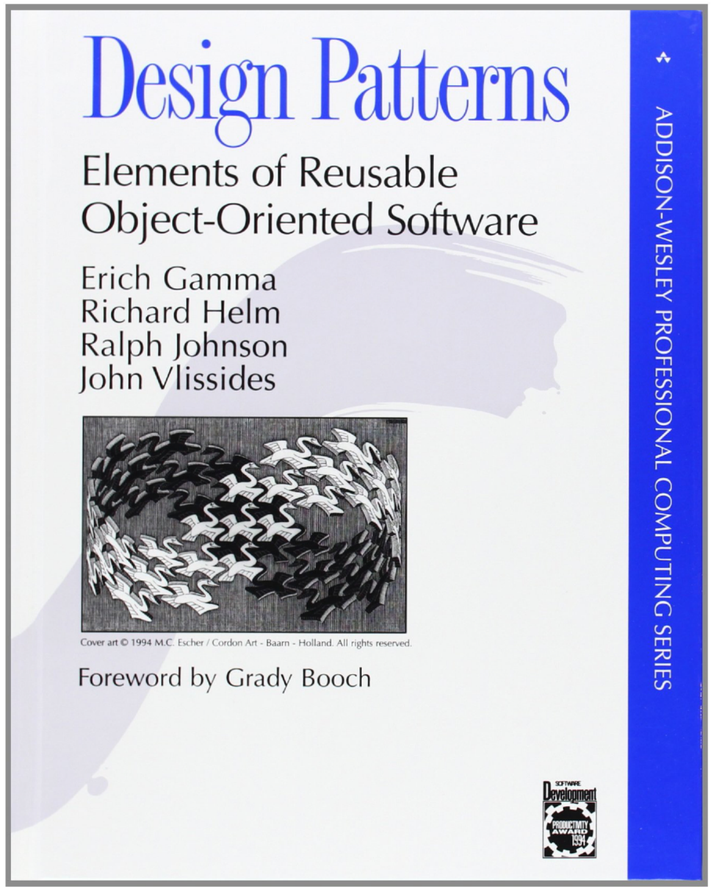
- Function-based 1D random walk:

```
import numpy as np
import numpy.random as nr

N = 20
p = 0.5
x0 = 0
_dx = lambda: -1 if nr.random()<p else 1

print x0 + np.cumsum([_dx() for n in range(N)])
```

- OO approach quite natural for *agent based* models
- Simple to reuse and extend
- Might introduce severe overhead

Further reading





Object oriented programming

P3 — Consider the script **`main_randomWalk.py`** that implements data-structures and functions for simulating a basic 1D random walk. *Wade* through the script!

Keep distinct aspects of your work separate: often it is a good idea to separate expensive simulations (in terms of CPU-time) from data-analysis.

(A) Consider the function **`main_displacement()`** in the script **`main_randomWalk.py`**. Amend the function so that it accumulates

$$x_n = \sum_{i=0}^{n-1} \Delta x_i \quad \text{and} \quad x_n^2 = \left[\sum_{i=0}^{n-1} \Delta x_i \right]^2 ,$$

i.e. the displacement and squared displacement of individual random walkers, respectively. The function saves the averaged values to a file called **`RW_displacement.dat.npz`**.

Object oriented programming

(B) Use the script **`figure_diffusionCoefficient.py`** to post process the data. It estimates and displays the diffusion coefficient from the data set generated in subproblem (A). Wade through that script also!

To verify your modelling approach consider the following exact results for the 1D random walk:

- The mean square displacement scales like

$$\langle x_N^2 \rangle = 4(1-p)p \; N \; \Delta x^2$$

- The diffusion coefficient is obtained as

$$D = \frac{1}{2} \lim_{N \to \infty} \left[\frac{\langle x_N \rangle^2}{N} \right]$$

Object oriented programming

P4 — Consider a function based or object-oriented approach to implement a simple application for a restricted random walk. Learn about the restricted random walk by reading the documentation of the function **`runUntilTrapped()`**.

(A) Consider the function **`main_restrictedRW()`** in the script **`main_randomWalk.py`**. Amend the function so that it accumulates the number of steps taken by each individual random walker until it got trapped.

(B) Use the script **`figure_meanFirstPassageTime.py`** to post process the data. It prepares a running average of the mean first passage time from your simulated data.

Exact result for the restricted 1D random walk: the mean number of steps until a walker gets trapped, i.e. the *mean first passage time*, reads

$$\tau = \frac{x_0(L - x_0)}{2D}$$

Object oriented programming

P5 — Use the *simulation method* for the 1D random walk as a tool for the *numerical analysis* of a differential equation.

(A) Consider the function **`main_endpointDistribution()`** in the script **`main_randomWalk.py`**. Set the simulation parameters for the 1D random walks and the parameters of the 1D diffusion equation solver so that both describe the same scenario. What do you *expect*? What do you *observe*?

Object oriented programming

P6 — Implement a symmetric 1D random walk. A symmetric 1D random walk takes a step to the left or right with equal probability *p*. With probability *1-2p* it stays put!

(A) *Override* the stepping-method of the 1D random walk class to yield a symmetric random walk.

If you have trouble designing an adequate stepping-method, consider the implementation:

```
def _dx():
    r = random()
    return -1*(r>1-p)+1*(r<p)
```

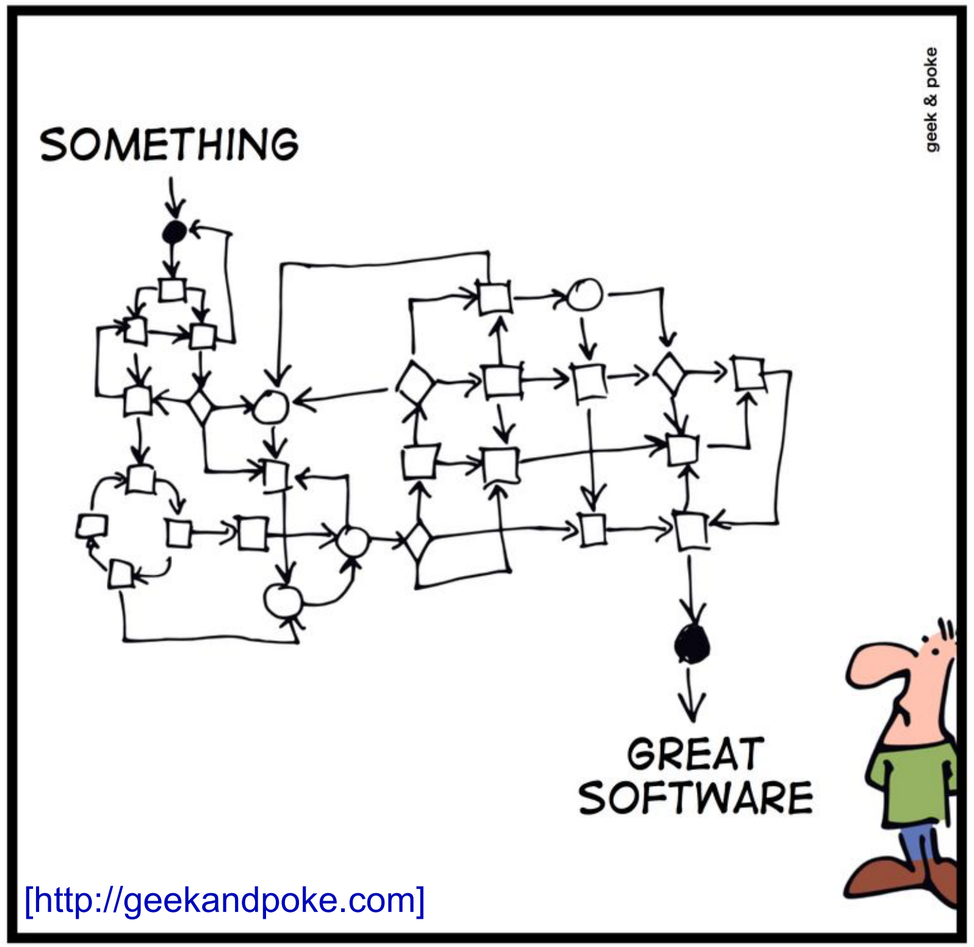
Figure out how the random variable works!

How to write python™ code?

Minimal set of guidelines to write **good** code:

1. **Write programs for people (not computers)**
   - prefer clarity over brevity
   - strive for clear and consistent code-style
2. **Document design and purpose**
   - comment interfaces and reasons
   - embed the documentation in the software
3. **Plan for mistakes — test**
   - development process: test early
   - software unit: test as soon as written
4. **Collaborate with other scientists**
   - use (test) code-reviews
   - incorporate feedback from your peers
5. **Follow a software development methodology**
   - agree on coding standards and respect them

[G. Wilson *et al.*, Best Practices for Scientific Computing, PLoS Biology (2014)]



Caution: don't over-obsess on software-engineering practices!

A well documented interface!