# *Developing large-scale Applications in Python*

## *Lessons learned from 10 years of Python Application Design*

UK Python Conference 2004
Oxford, United Kingdom

Marc-André Lemburg

EGENIX.COM Software GmbH
Germany

## Speaker Introduction: **Marc-André Lemburg**

- CEO eGenix.com Software GmbH, Germany

  – Founded in 2000

  – Core business: projects and products using Python and C

- Consultant

  – More than 20 years software experience

  – Diploma in Mathematics

  – Expert in Python, OOP, Web Technologies and Unicode

  – Python Core Developer

  – Python Software Foundation Board Member (02/04)

  – Contact: mal@egenix.com
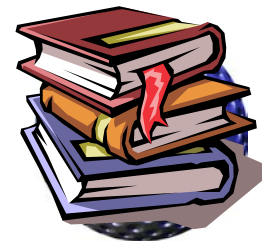
2

# Introduction

3

## Python2EE: **Large-scale applications**

- What is considered "large-scale" in Python ?

  - Server application: >30 thousand lines of Python code
  - Client application: >10 thousand lines of Python code

  - Third-Party code: >10 thousand lines of code
  - Typically a mix of Python code and C extensions

- Examples:

  - Zope / Plone
  - eGenix Application Server
  - eGenix projects: e.g. XML Database, Finance Applications

4

# Application Design

## Path to success: **Application Design**

- Python makes it very easy to write complex applications with very little code

  – It's easy to create bad designs fast

  – Rewriting code is fast as well

- Application design becomes the most important factor in Python projects

- Here's a cookbook approach to the problem…

6

## Breaking it down: **The Design Concept**

- Zen

    – Keep things as simple as possible, but not simpler (KISS).

    – There's beauty in design.

    – Before doing things twice, think twice.

    – If things start to pile up, management is needed.

    – If management doesn't help, decomposition is needed.

- Structured approach to application design

    – Divide et Impera (divide and conquer)

    – Lots and lots of experience ☺
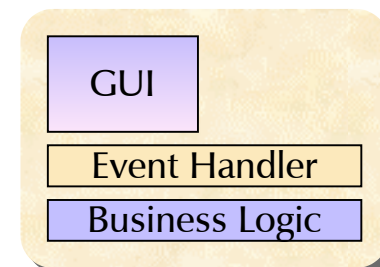
## First step: **Choose a suitable application model**

- ## Client-Server

  - Client application / Server application
  - Web client / Server application

    *Client*

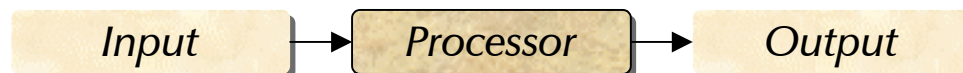    *Server*
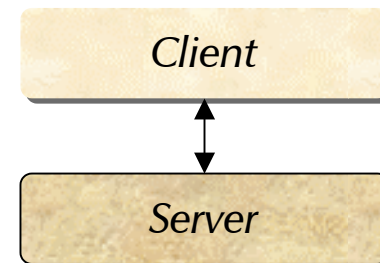
- ## Multi-threaded stand-alone
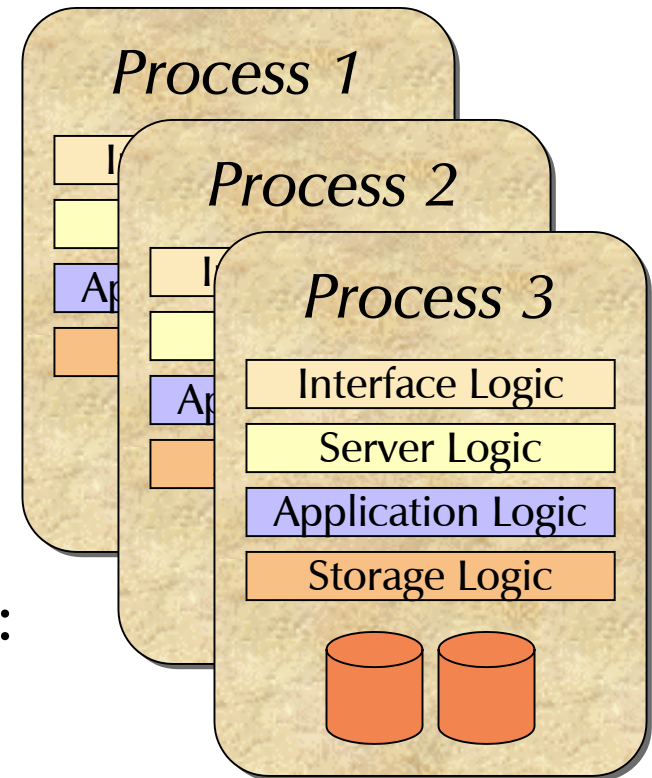
  - Stand-alone GUI application

    GUI

    Event Handler

    Business Logic

- ## Single process

  - Command-line application
  - Batch job application

    *Input* → *Processor* → *Output*

- ## etc.

≡GΞΝiX.COM

# Bytes in action: Requirements for request processing

- Identify the processing scheme:
  - Single process
  - Multiple processes
  - Multiple threads
  - Asynchronous processing
  - A mix of the above

- Identify the process/thread boundaries:
  - Which components (need to) share the same object space ?
  - Where is state kept ?
  - What defines an application instance ?

**Process 1**

**Process 2**

**Process 3**

| Interface Logic |
| Server Logic |
| Application Logic |
| Storage Logic |

## Looking closer: **Application layers**

- Every application can be divided into layers of functionality defined by the flow of data through the application

  - Top layer: interface to the outside world

  - Intermediate layers: administration and processing

  - Bottom layer: data storage

- Layers are usually easy to identify given the application model

  - … but often hard to design

*Application*

Interface Logic

Administration Logic

Application Logic
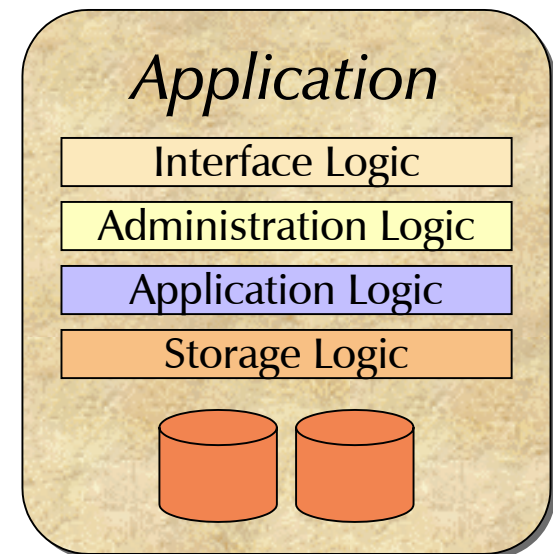
Storage Logic

# Vertical design: **Find the right application** layer model

- GUI / Application Logic / Storage Logic

- Network / Apache / SCGI / Server Logic / Application Logic / Storage Logic

- File I/O / Application Logic / Storage Logic

- Custom model

*Client*

Web Browser

*Server*

Interface Logic

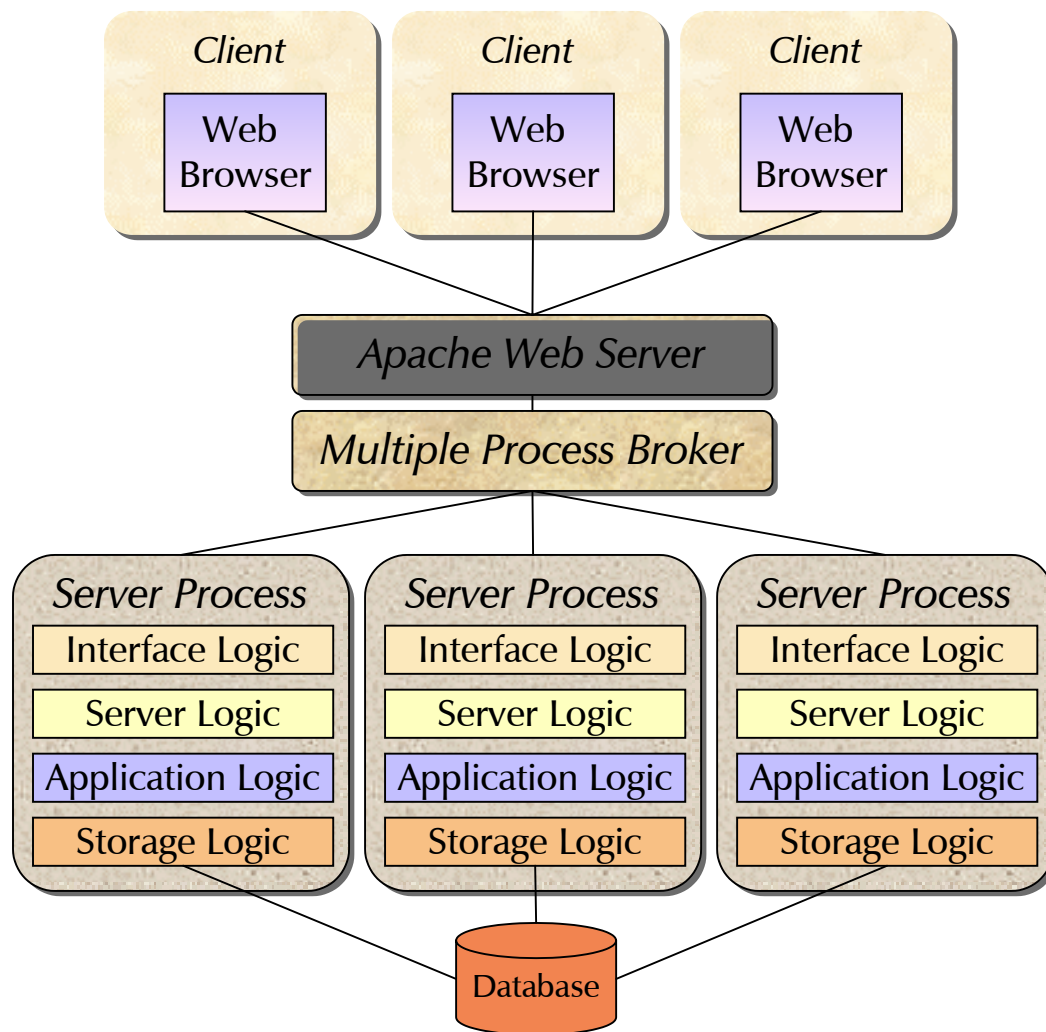Server Logic

Application Logic

Storage Logic

11
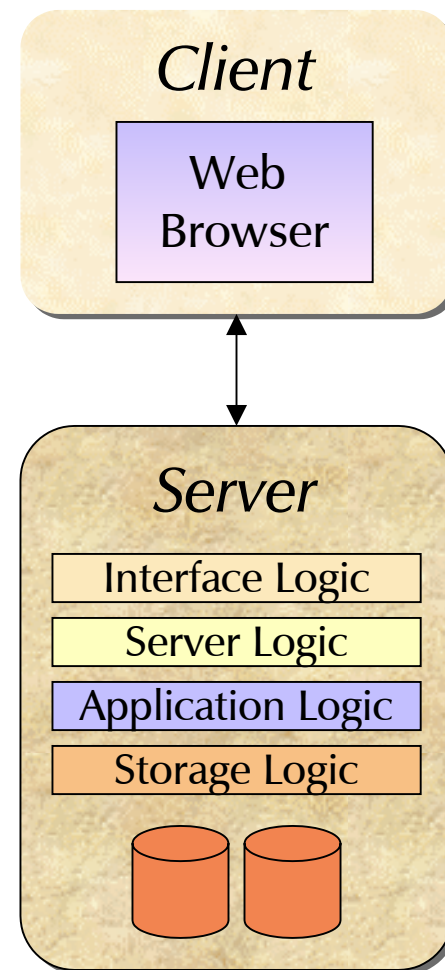
## Example: Web Client + Server Application

- Setup:

  – Client is a standard
  web-browser

  – Server needs to take a lot
  of load and will have to
  do all the calculation
  work

  – Server needs to be
  fail-safe

- Solution:

  – Multiple process model

  – Application server layers

# The next step: **Breaking layers into smaller pieces**

- Layers provide a data driven separation of functionality

- Problem:
  - The <span style="color:red">level of complexity is usually too high</span> to implement these in one piece of code

- Solution:
  - build layers using a set of <span style="color:blue">loosely coupled components</span>

13

# Plug & play: **Component** design

- Components should encapsulate <span style="color:red">higher level concepts</span> within the application, e.g.

    - provide the database interface
    - implement the user management
    - implement the session management
    - provide caching facilities
    - interface to external data sources
    - provide error handling facilities
    - enable logging management
    - etc.

## Advantages of components
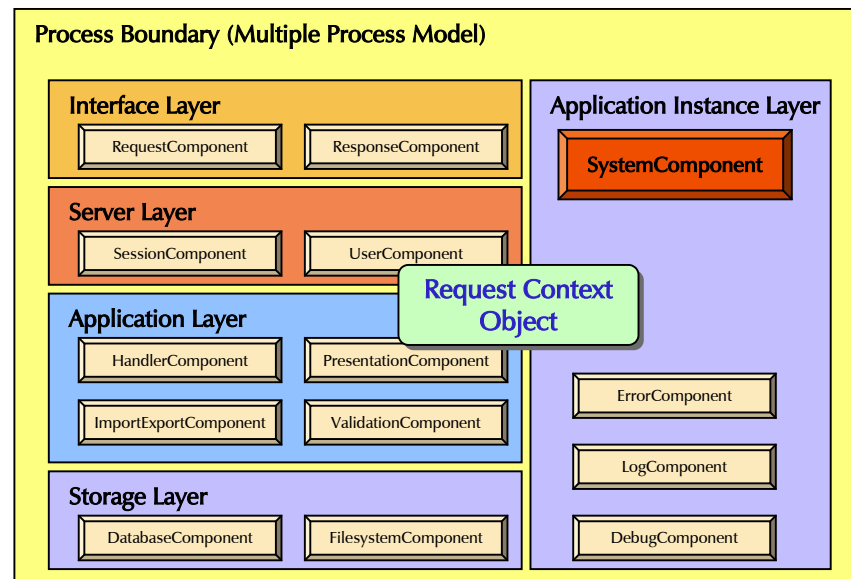
- Components provide <span style="color:red">independent building blocks</span> for the application

  - They should be <span style="color:blue">easily replaceable</span> to adapt the application to new requirements, e.g.
    - porting the application to a new database backend,
    - using a new authentication mechanism, etc.
    - ➢ If implemented correctly, they will allow switching to different processing model should the need arise

  - <span style="color:blue">Loose coupling</span> of the components make it possible to
    - refine the overall application design,
    - refactor parts of the layer logic or
    - add new layers
    - ➢ without having to rewrite large parts of the application code

15

EGENIX.COM

## Hitchhiker's guide to: **Component implementation**

- Each component is represented by a component object

- One system component object (representing the application instance):
  - All component objects are created and managed by the system object
  - Components can access each other through the system object (circular references !)

- Component interfaces must be simple and high-level enough to allow for loose coupling

  - Internal parts of the components are never accessed directly, only via the component interface

- Note: Component objects should never keep state across requests (ideally, they should be thread-safe)

## Horizontal design: **Split the layers into components**

- General approach:
  - One system component that manages the application instance
  - At least one component per layer

- Data management:
  - Global data is only used for configuration purposes
  - Components don't store per-request state !

  ➢ Per-request global data is stored and passed around via Request Context Objects

Process Boundary (Multiple Process Model)

Interface Layer
RequestComponent  ResponseComponent

Server Layer
SessionComponent  UserComponent

Application Layer
HandlerComponent  PresentationComponent
ImportExportComponent  ValidationComponent

Storage Layer
DatabaseComponent  FilesystemComponent

Application Instance Layer
SystemComponent

Request Context Object

ErrorComponent

LogComponent

DebugComponent

17

# The big picture: Layers and components

**Process Boundary (Multiple Process Model)**

### Interface Layer

| RequestComponent | ResponseComponent |

### Server Layer

| SessionComponent | UserComponent |

### Application Layer

| HandlerComponent | PresentationComponent |
| ImportExportComponent | ValidationComponent |

### Storage Layer

| DatabaseComponent | FilesystemComponent |

### Application Instance Layer

| SystemComponent |

All Component Objects are connected to the SystemComponent object

| ErrorComponent |

| LogComponent |

| DebugComponent |

18

# Teams at work: **Management objects**

- Management objects
  - work on or with groups of data/task objects
  - help simplify component object implementations

- Tasks:
  - interaction with multiple objects
  - I/O on collections of objects
  - delegating work to other management objects
  - interfacing to component objects
  - etc.

- The distinction between management objects and component objects is not always clear
  - If in doubt, use a component object that proxies to a management object

# Drill-down: Management objects at work…



20

EGENIX.COM
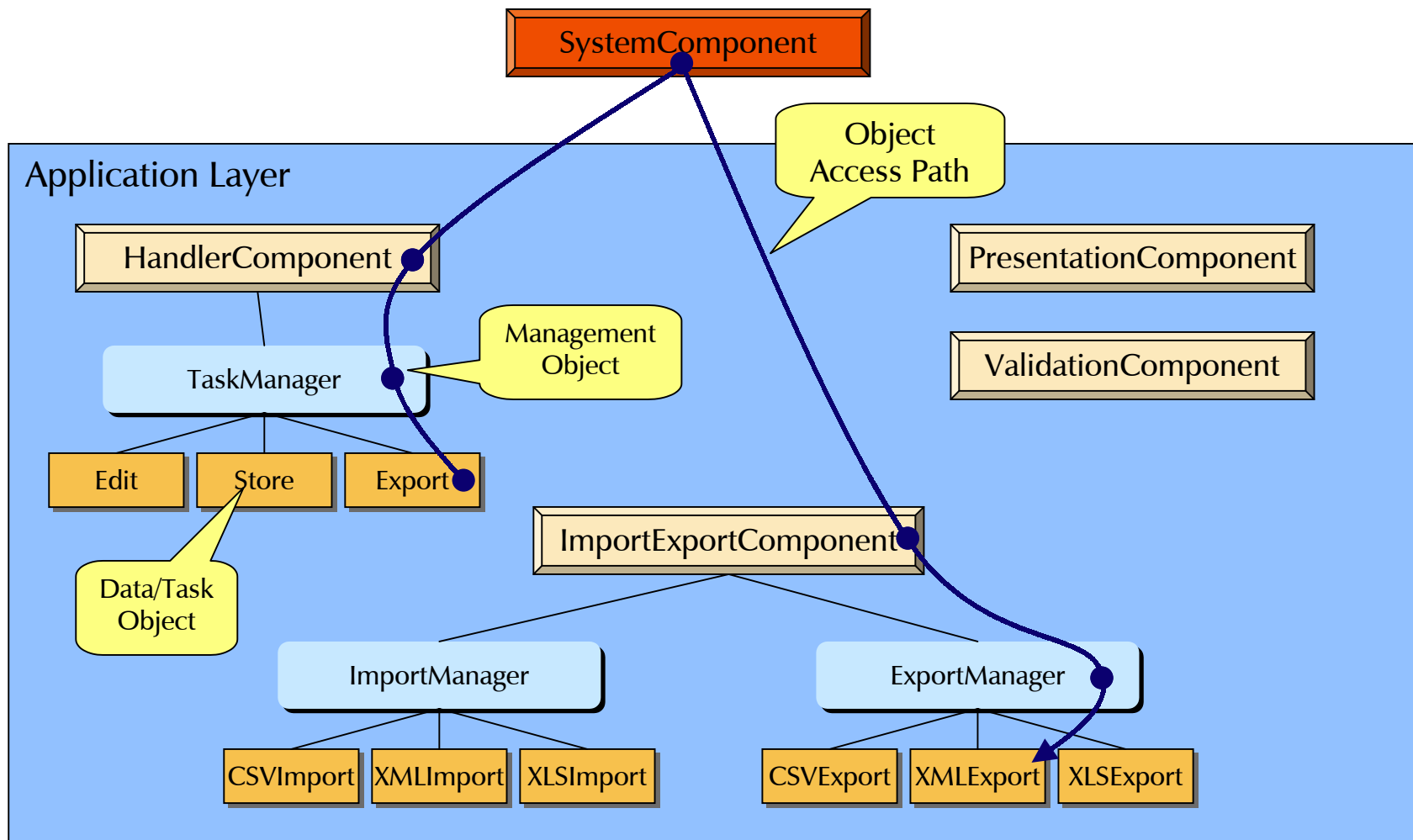
## Managing responsibilities: **Who's in charge ?**

- Use management objects to work on collections of data/task objects

  – avoid direct interfacing between the data/task objects

- Rely on component objects to provide additional facilities to management objects

  – rather then coding them into the management objects

➢ Never mix responsibilities

ΞGΞNIX.COM

## Reality check: **What have we learned ?**

- Application design is in many ways like structuring a company:

  - Divisions need to be set up (component objects)
  - Responsibilities need to be defined (management vs. data/task objects)
  - Processes need to be defined (component/management object APIs)

- Applications work in many ways like companies:

  - Customer interaction (user interface)
  - Information flow (application interface)
  - Decision process (business logic)
  - Accounting and data keeping (storage interface)

=GENIX.COM

# At work...

1. Introduction

2. Application Design

3. At work...

4. Discussion

## Before you start: **Structuring your modules**

- First some notes on the import statement:

  - Keep import dependencies low;
    avoid "from … import *"

  - Always use absolute import paths
    (defeats pickle problems among other things)

  - Always layout your application modules using Python packages

  - Import loops can be nasty;
    import on demand can sometimes help

## Some guidelines: **Finding the right package structure**

- Use <span style="color:red">one module</span> per

  - management/component class

  - group of object classes
    managed by the same management class

  - ➢ keep modules small;
    if in doubt, split at class boundaries

- Group components and associated
  management modules in
  Python packages (directories)

- Use the application model as basis for
  the package layout

25

ΞGENIX.COM

# Finding the right mix: **Data, classes and methods**

- Use data objects for data encapsulation…
  - instead of simple types (tuples, lists, dictionaries, etc.)

- Use methods even for simple tasks…
  - but don't make them too simple

- Use method groups for more complex tasks
  - e.g. to implement a storage query interface

- Use mix-in classes if method groups can be deployed in more than class context
  - If you need to write the same logic twice, think about creating a mix-in class to encapsulate it, or put it on a base class
  - Avoid using mix-in classes if only one class makes use of them

26

## Make mistakes… and learn from them: **Refactoring**

- If an implementation gets too complicated, sit down and reconsider the design…

  - often enough a small change in the way objects interact can do wonders

- Be daring when it comes to rewriting larger parts of code !

- It sometimes takes more than just a few changes to get a design right

- A good design will always be faster to implement than trying to fix a broken one
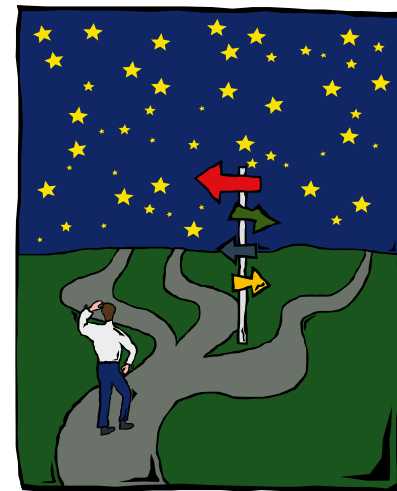
## Often forgotten: **Documentation**

- Always document the code
  that you write !

- Use doc-strings, inline comments and
  block logical units using empty lines…

  – doc-strings represent your method's
    contracts with the outside world

- Document the intent of the methods, classes
  and logical code units…

  – not only their interface

- Use descriptive identifier names…

  – even if they take longer to type

28
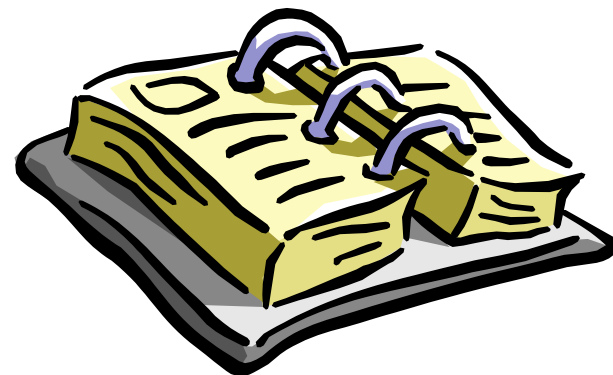
## Five minutes that make a difference: **Quality Assurance**

- Use extreme programming techniques whenever possible:

  - Always read the code top to bottom after you have made changes or added something new to it
  - Try to follow the flow of information in your mind (before actually running the code)
  - Write unit tests for the code and/or test it until everything works as advertised in the doc-strings

- Typos can easily go unnoticed in Python: use the editor's auto-completion function as often as possible

- Use tools like PyChecker to find hidden typos and possibly bugs

- Always test code before committing it to the software repository

29

# EGENIX.COM

# Discussion

1. Introduction

2. Application Design

3. At work...

4. Discussion

# Developing large-scale applications in Python

- Questions

  - Has anyone worked on large-scale Python applications ?

  - What tools / features are (still) missing in the tool chain ?

  - Would you be prepared to pay for components or frameworks ?

31

# And finally...



Thank you for your time.

## Contact

eGenix.com Software, Skills and Services GmbH

Marc-André Lemburg

Pastor-Löh-Str. 48

D-40764 Langenfeld

Germany

| | |
|---|---|
| eMail: | mal@egenix.com |
| Phone: | +49 211 9304112 |
| Fax: | +49 211 3005250 |
| Web: | http://www.egenix.com/ |

33