

# Plone 3 Techniques

## Plone Symposium East 2008

**Author:** Joel Burton <[joel@joelburton.com](mailto:joel@joelburton.com)>  
**Copyright:** Copyright 2002-2008 Joel Burton  
**Covering:** Plone 3.0 or newer  
**Notice:** Distribution outside of course prohibited.  
**Info:** Offered through PloneBootcamps:  
[www.plonebootcamps.com](http://www.plonebootcamps.com)

## Contents

<b>1</b>	<b>Welcome to Class!</b>	<b>16</b>
1.1	Class . . . . .	17
1.1.1	Topics . . . . .	17
1.1.2	Topics (2) . . . . .	17
1.1.3	Topics (3) . . . . .	17
1.1.4	What (Mostly) Hasn't Changed . . . . .	18
1.2	How Things Work . . . . .	19
1.2.1	Advice from Past Students . . . . .	19
1.2.2	Pace of Class . . . . .	19
1.3	Stereograms . . . . .	20
<b>2</b>	<b>Plone 3 Content Management</b>	<b>21</b>
2.1	Overview . . . . .	22
2.2	Click to Edit . . . . .	23
2.2.1	Click to Edit . . . . .	23
2.2.2	Click to Edit Caveats . . . . .	23
2.2.3	Oh, the Humanity! . . . . .	23
2.3	New Schematas . . . . .	25
2.4	Renaming . . . . .	26
2.4.1	Keywords . . . . .	26
2.4.2	Smart Folders . . . . .	26
2.5	Sharing Tab . . . . .	27
2.5.1	Sharing Tab . . . . .	27
2.5.2	Changing Ownership . . . . .	27
2.5.3	Roles to Share . . . . .	27
2.5.4	Roles to Share II . . . . .	28
2.5.5	Local Role Pattern . . . . .	28
2.5.6	Local Role Blacklisting . . . . .	29
2.5.7	Logged-in Users Group . . . . .	29
2.5.8	File Fields Searched . . . . .	29
2.5.9	File Fields Search Setup . . . . .	30
2.5.10	Link Monitoring . . . . .	30

<b>3</b>	<b>Plone 3 Workflows</b>	<b>31</b>
3.1	Overview . . . . .	32
3.2	New Workflows . . . . .	33
3.2.1	Standard Plone Workflow . . . . .	33
3.2.2	Simple Publication Workflow . . . . .	33
3.2.3	One State Workflow . . . . .	34
3.2.4	Plone Workflow . . . . .	34
3.2.5	Plone Workflow . . . . .	34
3.2.6	Folder Workflow . . . . .	35
3.2.7	Intranet Workflow . . . . .	35
3.2.8	Intranet Workflow . . . . .	36
3.2.9	Intranet Folder Workflow . . . . .	36
3.3	Workflow Changes . . . . .	37
3.3.1	Files & Images . . . . .	37
3.3.2	Default Pages . . . . .	37
3.4	Placeful Workflow . . . . .	38
3.4.1	About Placeful Workflow . . . . .	38
3.4.2	Using Placeful Workflow . . . . .	38
3.4.3	Establishing Placeful Workflow . . . . .	38
3.4.4	Advantages of Placeful Workflow . . . . .	39
3.5	Customizing Workflows . . . . .	40
3.5.1	Customizing Workflows . . . . .	40
3.5.2	Transition Action . . . . .	40
3.5.3	Transition Action (2) . . . . .	40
<b>4</b>	<b>Versioning</b>	<b>41</b>
4.1	Overview . . . . .	42
4.2	Versioning . . . . .	43
4.2.1	Versioning . . . . .	43
4.2.2	Features . . . . .	43
4.2.3	Versioned Types . . . . .	43
4.2.4	Purging . . . . .	44

4.2.5	Staging . . . . .	44
4.2.6	Using Iterate . . . . .	44
4.2.7	Using Iterate II . . . . .	45
4.2.8	Using Iterate III . . . . .	45
4.2.9	Iterate Missing Features . . . . .	45
4.2.10	Iterate Security . . . . .	45
4.3	Plone 3 Practices . . . . .	46
<b>5</b>	<b>Plone 3 Portlets</b>	<b>47</b>
5.1	Adding Portlets . . . . .	48
5.1.1	Adding Portlets . . . . .	48
5.1.2	Type-Specific Portlets . . . . .	48
5.1.3	Group-Specific Portlets . . . . .	48
5.1.4	Blocking Portlets . . . . .	48
5.1.5	Useful Add-On Portlets . . . . .	49
5.2	Dashboard . . . . .	50
<b>6</b>	<b>Content Rules</b>	<b>51</b>
6.1	Overview . . . . .	52
6.1.1	Overview . . . . .	52
6.1.2	Adding a Rule . . . . .	52
6.1.3	Conditions . . . . .	53
6.1.4	Actions . . . . .	53
6.2	Limitations . . . . .	54
<b>7</b>	<b>Plone 3 Skinning</b>	<b>55</b>
7.1	Overview . . . . .	56
7.1.1	Overview . . . . .	56
7.1.2	Zope3 Templates . . . . .	56
7.2	Customerize . . . . .	57
7.2.1	Customerize . . . . .	57
7.2.2	Customerize (2) . . . . .	57

<b>8</b>	<b>Plone 3 Archetypes</b>	<b>58</b>
8.1	Overview . . . . .	59
8.2	Archetypes . . . . .	60
8.2.1	Archetypes . . . . .	60
8.2.2	Converting UML . . . . .	60
8.2.3	Permissions . . . . .	60
8.2.4	Skinning . . . . .	61
8.2.5	base_view Example . . . . .	61
<b>9</b>	<b>GenericSetup</b>	<b>62</b>
9.1	Introduction . . . . .	63
9.2	Generic Setup . . . . .	64
9.2.1	About Generic Setup . . . . .	64
9.2.2	Profiles . . . . .	64
9.2.3	Extension Profiles . . . . .	64
9.2.4	Snapshots . . . . .	65
9.2.5	Snapshots (2) . . . . .	65
9.2.6	Export . . . . .	65
9.2.7	Factoring What You Need . . . . .	65
9.2.8	Sample Properties.xml . . . . .	66
9.2.9	Sample Workflows.xml . . . . .	66
9.2.10	Import . . . . .	66
9.2.11	Using with Products . . . . .	67
9.2.12	GS and ZCML . . . . .	67
9.2.13	GS and QuickInstaller . . . . .	68
9.2.14	Getting GS w/o ZCML . . . . .	68
9.2.15	GS and Python Code . . . . .	68
9.2.16	GS and Python Code II . . . . .	69
9.2.17	GS and Python Code III . . . . .	69
9.3	Exercises . . . . .	70
9.3.1	Exercises . . . . .	70
9.3.2	Exercises Answers #1 . . . . .	70

9.3.3	Exercises Answers #2 . . . . .	71
9.3.4	Exercises Answers #3 . . . . .	71
9.3.5	Exercises Answers #4 . . . . .	71
<b>10</b>	<b>Buildout Basics</b>	<b>73</b>
10.1	Overview . . . . .	74
10.2	Buildout . . . . .	75
10.2.1	Buildout . . . . .	75
10.2.2	Who Is It For? . . . . .	75
10.2.3	Idea . . . . .	75
10.2.4	Getting Started . . . . .	76
10.2.5	Eggs . . . . .	76
10.2.6	Eggs 2 . . . . .	76
10.2.7	Challenges . . . . .	76
10.2.8	But ... . . . .	77
10.2.9	Does this Replace My Installer? . . . . .	77
10.2.10	Eggs without Buildout . . . . .	77
10.2.11	Eggs without Buildout 2 . . . . .	78
10.2.12	Eggs without Buildout 3 . . . . .	78
<b>11</b>	<b>Plone 3 and Python</b>	<b>79</b>
11.1	Overview . . . . .	80
11.2	Classes . . . . .	81
11.2.1	Classes . . . . .	81
11.2.2	Classes 2 . . . . .	81
11.2.3	Classes 3 . . . . .	81
11.2.4	Decorators . . . . .	82
11.2.5	Decorator Example . . . . .	82
11.2.6	Decorators . . . . .	82
11.2.7	Unicode . . . . .	83

<b>12 Interfaces and ZCML</b>	<b>84</b>
12.1 Overview . . . . .	85
12.2 What Is Zope 3 . . . . .	86
12.2.1 Benefits of Zope 3 . . . . .	86
12.2.2 Downsides of Zope 3 . . . . .	86
12.3 Interfaces . . . . .	87
12.3.1 Interfaces . . . . .	87
12.3.2 Student . . . . .	87
12.3.3 Student Interface . . . . .	88
12.3.4 Student Implementation . . . . .	88
12.3.5 Advantages of Interfaces . . . . .	89
12.3.6 Disadvantages of Interfaces . . . . .	89
12.4 Marker Interfaces . . . . .	90
12.4.1 “Marker Interfaces” . . . . .	90
12.4.2 Zope 2-Style Markers . . . . .	90
12.4.3 Zope 3-Style Markers . . . . .	90
12.4.4 Zope 3-Style Markers II . . . . .	91
12.5 ZCML . . . . .	92
12.5.1 ZCML Intro . . . . .	92
12.5.2 Example ZCML . . . . .	92
12.5.3 Example ZCML: Five . . . . .	92
12.5.4 Marking Class with Iface . . . . .	93
12.5.5 Finding Things in ZCML . . . . .	93
12.5.6 Marker Interface TTW . . . . .	93
12.6 Plone 3 Practices . . . . .	94
12.6.1 Hiding From Breadcrumbs . . . . .	94
12.6.2 Hiding From Breadcrumbs II . . . . .	94
12.7 ZCML Organization . . . . .	95
12.7.1 ZCML Organization . . . . .	95
12.7.2 ZCML Organization II . . . . .	95
12.7.3 ZCML Organization III . . . . .	95
12.8 Exercises . . . . .	96

---

12.8.1 Exercises . . . . .	96
12.8.2 Exercises Answers #1 . . . . .	96
<b>13 Utilities and Adapters</b>	<b>97</b>
13.1 Overview . . . . .	98
13.2 Utilities . . . . .	99
13.2.1 Utilities . . . . .	99
13.2.2 Making a Utility . . . . .	99
13.2.3 Making a Utility II . . . . .	99
13.2.4 Making a Utility III . . . . .	100
13.2.5 Using Utilities . . . . .	100
13.3 Named Utilities . . . . .	101
13.3.1 Named Utilities . . . . .	101
13.3.2 Making a Named Utility . . . . .	101
13.3.3 Using a Named Utility . . . . .	101
13.4 Overriding ZCML . . . . .	102
13.4.1 Conflicting ZCML . . . . .	102
13.4.2 Overriding ZCML . . . . .	102
13.5 Products Versus Packages . . . . .	103
13.5.1 Products? Packages? . . . . .	103
13.5.2 Products? Packages? II . . . . .	103
13.5.3 ZCML Slugs . . . . .	103
13.5.4 Sample ZCML Slug . . . . .	104
13.6 Adapters . . . . .	105
13.6.1 Adapters . . . . .	105
13.6.2 Subclassing . . . . .	105
13.6.3 Subclassing Problems . . . . .	105
13.6.4 Adapter Benefit . . . . .	106
13.6.5 Adapter Example . . . . .	106
13.6.6 Adapter Example . . . . .	106
13.6.7 Adapter Example . . . . .	106
13.6.8 Adapter Example . . . . .	107

13.6.9	Example ZCML . . . . .	107
13.6.10	Plone 3 Techniques . . . . .	107
13.7	Exercises . . . . .	108
13.7.1	Exercises . . . . .	108
13.7.2	Exercises Answers #1 . . . . .	108
13.7.3	3exercise Answers #2 . . . . .	108
13.7.4	Exercise Answers #3 . . . . .	109
13.7.5	Exercise Answers #4 . . . . .	109
13.7.6	Exercise Answers #4 . . . . .	109
<b>14</b>	<b>Views and View Classes</b>	<b>110</b>
14.1	Overview . . . . .	111
14.2	Views . . . . .	112
14.2.1	Views . . . . .	112
14.2.2	Sample View . . . . .	112
14.2.3	Sample View II . . . . .	113
14.2.4	Sample View III . . . . .	113
14.2.5	Sample View IV . . . . .	113
14.3	Permissions . . . . .	114
14.3.1	Permissions . . . . .	114
14.3.2	New Permissions . . . . .	114
14.3.3	Significant Security Change . . . . .	114
14.3.4	The Goggles . . . . .	115
14.3.5	View Classes . . . . .	115
14.3.6	Zope2-style "View Class" . . . . .	115
14.3.7	Zope2-style "View Class" . . . . .	115
14.3.8	Zope2-style Overview . . . . .	116
14.3.9	Zope3-style View Class . . . . .	116
14.3.10	Zope3-style View Class . . . . .	116
14.3.11	Using Zope3 View Class . . . . .	117
14.3.12	Binding Class and View Together . . . . .	117
14.3.13	Binding Class and View Together II . . . . .	117

14.4 Plone 3 Practices . . . . .	118
14.4.1 “Browser” Package . . . . .	118
14.4.2 “Templates” Directory . . . . .	118
<b>15 Viewlets</b>	<b>119</b>
15.1 Overview . . . . .	120
15.2 Viewlets . . . . .	121
15.2.1 Viewlets . . . . .	121
15.2.2 Re-Arranging Viewlets . . . . .	121
15.2.3 A Viewlet . . . . .	121
15.2.4 Viewlet ZCML Registration . . . . .	122
15.2.5 A Viewlet with Logic . . . . .	122
15.2.6 Viewlet Class . . . . .	122
15.2.7 Viewlet ZCML Registration . . . . .	123
15.3 Re-Arranging and Hiding . . . . .	124
15.3.1 Rearranging and Hiding . . . . .	124
15.3.2 Specifying Viewlet Order . . . . .	124
15.3.3 Removing Viewlets . . . . .	124
15.3.4 Hiding Viewlets . . . . .	125
15.3.5 Inserting in Order . . . . .	125
15.3.6 Viewlet Managers . . . . .	125
15.3.7 New Viewlet Manager . . . . .	126
15.3.8 Adding Viewlets to Manager . . . . .	126
15.4 Layers . . . . .	127
15.4.1 Layers . . . . .	127
15.4.2 Making a Layer . . . . .	127
15.4.3 Binding Viewlets to Theme . . . . .	127
<b>16 Plone 3 Portlets</b>	<b>129</b>
16.1 Overview . . . . .	130
16.2 Classic Portlets . . . . .	131
16.2.1 Classic Portlets . . . . .	131
16.2.2 Sample Classic Portlet . . . . .	131

---

16.3	New-Style Portlets . . . . .	132
16.3.1	New-Style Portlet . . . . .	132
16.3.2	New Mission Portlet . . . . .	132
16.3.3	New Mission Portlet II . . . . .	133
16.3.4	New Mission Portlet III . . . . .	133
16.3.5	New Mission Portlet IV . . . . .	134
16.3.6	New Mission Portlet V . . . . .	134
16.3.7	New Mission Portlet VI . . . . .	135
16.3.8	New Mission Portlet VII . . . . .	135
16.3.9	New Mission Portlet VIII . . . . .	136
16.3.10	Classic or New Style . . . . .	136
16.3.11	Classic or New Style II . . . . .	136
16.3.12	Classic or New Style III . . . . .	137
<b>17</b>	<b>Formlib Intro</b>	<b>138</b>
17.1	Overview . . . . .	139
17.1.1	Overview . . . . .	139
17.1.2	Concepts . . . . .	139
17.2	Example . . . . .	140
17.2.1	Example 1 . . . . .	140
17.2.2	Implementation Class . . . . .	140
17.2.3	Example 2 . . . . .	141
17.2.4	Registering with ZCML . . . . .	141
17.2.5	Example 3 . . . . .	141
17.2.6	Common Field Types: String . . . . .	142
17.2.7	Common Field Types: Numbers . . . . .	142
17.2.8	Common Field Types: Dates . . . . .	142
17.2.9	Common Field Types: Other . . . . .	143
17.2.10	Common Field Paramters . . . . .	143
17.3	Road Ahead . . . . .	144

<b>18 KSS Intro</b>	<b>145</b>
18.1 Overview . . . . .	146
18.1.1 Overview . . . . .	146
18.1.2 What is KSS? . . . . .	146
18.2 Client-Side . . . . .	147
18.2.1 Calculator Form . . . . .	147
18.2.2 KSS . . . . .	147
18.2.3 KSS Syntax . . . . .	148
18.2.4 KSS 2 . . . . .	148
18.3 Server-Side . . . . .	149
18.3.1 KSS . . . . .	149
18.3.2 Script . . . . .	149
18.3.3 Alternate Script . . . . .	150
18.3.4 Alternate Script 2 . . . . .	150
18.4 KSS Cheat Sheet . . . . .	151
18.4.1 Actions: Changing HTML . . . . .	151
18.4.2 Actions: Attributes . . . . .	151
18.4.3 Actions: CSS Classes . . . . .	152
18.4.4 Actions: Form Elements . . . . .	152
18.4.5 Actions: Debugging . . . . .	152
18.4.6 Parameter Providers: Forms . . . . .	153
18.4.7 Parameter Providers: Content . . . . .	153
18.4.8 Command Sets: Core . . . . .	153
18.4.9 Commands Sets: Zope + Plone . . . . .	154
18.4.10 Debugging KSS . . . . .	154
<b>19 Pluggable Auth Service</b>	<b>155</b>
19.1 Concepts . . . . .	156
19.1.1 Plone PAS . . . . .	156
19.1.2 PAS Concepts . . . . .	156
19.1.3 PAS Plugins . . . . .	157
19.2 Only In-Network . . . . .	158

19.2.1	Only In-Network Concept . . . . .	158
19.2.2	In-Network Script . . . . .	158
19.2.3	Hide Login Form . . . . .	158
19.2.4	Script PAS Plugins . . . . .	158
19.2.5	Learning the API . . . . .	159
19.2.6	Refuse to Challenge . . . . .	159
19.2.7	Making it Work . . . . .	159
19.2.8	Refusing to Authenticate . . . . .	160
19.2.9	Making it Work . . . . .	160
19.2.10	Testing it Out . . . . .	160
19.3	Free Role . . . . .	161
19.3.1	Free Role Concept . . . . .	161
19.3.2	New Plugin . . . . .	161
19.4	On-Disk Plugins . . . . .	162
19.4.1	On-Disk Plugins . . . . .	162
19.4.2	On-Disk Plugins Details . . . . .	162
19.5	Exercises . . . . .	163
19.5.1	Exercises . . . . .	163
19.5.2	Excercise Answers . . . . .	163
<b>20</b>	<b>Plone 3 Fixes</b>	<b>164</b>
20.1	CM Fixes . . . . .	165
20.1.1	Turn Off Inline Editing . . . . .	165
20.1.2	Multipage Wizards . . . . .	165
20.1.3	Multipage Wizard Fix . . . . .	165
20.2	Security Fixes . . . . .	166
20.2.1	Adding Roles to Sharing . . . . .	166
20.2.2	Adding Roles to Sharing . . . . .	166
20.2.3	Adding Roles to Sharing II . . . . .	167
20.2.4	Old-Style Messages . . . . .	167
20.2.5	New-Style Messages . . . . .	167

<b>21 Building Content Rules</b>	<b>168</b>
21.1 Overview . . . . .	169
21.2 Building an Action . . . . .	170
21.2.1 Action Overview . . . . .	170
21.2.2 Interface . . . . .	170
21.2.3 Action . . . . .	171
21.2.4 Executor . . . . .	171
21.2.5 Add Form . . . . .	172
21.2.6 Edit Form . . . . .	172
21.2.7 ZCML Wiring . . . . .	172
21.2.8 ZCML Wiring II . . . . .	173
21.2.9 ZCML Wiring III . . . . .	173
<b>22 Internationalization</b>	<b>174</b>
22.1 Concepts . . . . .	175
22.2 Template i18n . . . . .	176
22.2.1 Template i18n . . . . .	176
22.2.2 Your Template . . . . .	176
22.2.3 Your Template: Need to Do . . . . .	177
22.2.4 Case 1: “Welcome to Plone” . . . . .	177
22.2.5 Case 1: “Welcome...” (2) . . . . .	177
22.2.6 Case 1: “Welcome...” (3) . . . . .	178
22.2.7 Case 2: Alt Text of Image . . . . .	178
22.2.8 Case 2: Alt Text (2) . . . . .	178
22.2.9 Case 2: Alt Text (2) . . . . .	179
22.2.10 Case 3: Dynamic Content . . . . .	179
22.2.11 Case 3: Dynamic Content (2) . . . . .	179
22.2.12 Case 3: Dynamic Content (3) . . . . .	180
22.2.13 Case 3: Dynamic Content (4) . . . . .	180
22.2.14 Case 4: Combining Ideas . . . . .	181
22.2.15 i18n Domain . . . . .	181
22.2.16 Complete Template . . . . .	181

---

22.3	Translating . . . . .	182
22.3.1	Creating Template . . . . .	182
22.3.2	POT File . . . . .	182
22.3.3	POTs and POs . . . . .	183
22.3.4	PO File Translate . . . . .	183
22.3.5	Help for Translators . . . . .	184
22.4	Translating Scripts . . . . .	185
22.5	Translating Content . . . . .	186
22.5.1	Translating Content . . . . .	186
22.5.2	LinguaPlone . . . . .	186
22.5.3	LinguaPlone Options . . . . .	186
22.5.4	Language-Independent . . . . .	187
<b>23</b>	<b>Unit Tests</b>	<b>188</b>
23.1	Overview . . . . .	189
23.2	Running Unit Tests . . . . .	190
23.3	Writing Unit Tests . . . . .	191
23.3.1	Setup . . . . .	191
23.3.2	Base Class . . . . .	191
23.3.3	Sample Test . . . . .	192
23.3.4	Test Suite . . . . .	192
23.3.5	Scaffolding . . . . .	192
23.3.6	Test Functions . . . . .	193
23.3.7	Helper Attributes . . . . .	193
23.3.8	Helper Methods . . . . .	193
23.4	DocTests . . . . .	194
23.4.1	About DocTests . . . . .	194
23.4.2	Sample DocTest . . . . .	194
23.4.3	DocTest . . . . .	195
23.4.4	DocTest (2) . . . . .	195
23.4.5	DocTest Tips . . . . .	195
23.5	Road Ahead . . . . .	196

# 1 Welcome to Class!

## 1.1 Class

---

### 1.1.1 Topics

- Content management changes
    - Workflow
    - Versioning
    - Portlets
    - Control Rules
- 

### 1.1.2 Topics (2)

- Integrator changes
    - Archetypes
    - Generic Setup
- 

### 1.1.3 Topics (3)

- Developer changes
  - Intro to Zope 3
  - Views
  - Viewlets
  - Building portlets / content rules
  - PAS

### 1.1.4 What (Mostly) Hasn't Changed

- Archetypes
- “Theming” (CSS)
- Testing
  - Selenium or unit tests
- Deployment
- Relational databases

## 1.2 How Things Work

---

### 1.2.1 Advice from Past Students

- 8 hours is a long day
    - Get some rest
    - Drink lots of water
  - Work on the exercises
  - Make notes in Acrobat reader
- 

### 1.2.2 Pace of Class

- Starts off gently, gets a bit faster
  - Please feel free to give me feedback
- If you know the answer, let me know!
  - ... if you're lost, tell me that, too

## 1.3 Stereograms

### Stereograms

- I feel comfortable...
- ... building a Plone 2.5 site
- ... building a Plone 2.5 product
- ... writing medium-complex Python code

## **2 Plone 3 Content Management**

## 2.1 Overview

### Overview

- Click-to-Edit
- New Schematas
- Renaming of Concepts

## 2.2 Click to Edit

---

### 2.2.1 Click to Edit

- View screens for content allow click-to-edit
    - Excellent for power-users doing fast edits
    - Sometimes confusing for novice users
      - \* Easy to trigger unexpected errors
    - Powered by KSS and async JavaScript (AJAX)
- 

### 2.2.2 Click to Edit Caveats

- Only works with
    - Uncustomized Archetype views
    - “Plone 3-ified” view templates
      - \* Since practically no one did things to anticipate this
  - Not (yet) supported for all AT field types
    - Missing for `float`, `lines`, and others
    - Not supported for add-on field types
- 

### 2.2.3 Oh, the Humanity!

- Early research shows confusing for end-users
  - May get better as it gets refined
  - Can turn off by turning off `.kss` in `portal_kss`
    - \* This loses live validation on edit form, though
    - \* Refactoring may allow this to be easier in future



## 2.3 New Schematas

### New Schematas

- Edit forms now use AJAX-y schemata tabs
  - Slower to load initial edit form, but must faster to switch!

## 2.4 Renaming

---

### 2.4.1 Keywords

- Now called “Categories”
    - May not be as intuitive for users
    - “Categories” can suggest a fixed taxonomy
- 

### 2.4.2 Smart Folders

- Now called “Collections”
  - May be a very helpful change (“folder” confused people)

## 2.5 Sharing Tab

---

### 2.5.1 Sharing Tab

- Radically simplified
    - Still exposes same functionality & reporting
    - No longer shows all roles
      - \* Only those “appropriate” for sharing (no Manager, Owner, etc)
    - Re-labels roles as capabilities (“Contributor” == “Can add”)
- 

### 2.5.2 Changing Ownership

- Link to change ownership of content item gone
    - Probably unintentional; useful for many sites
    - Feature still works but not exposed in UI:  
`http://path/to/content/ownership_form`
- 

### 2.5.3 Roles to Share

- **Can view** (Reader)
  - Can view (even when private/pending)
- **Can edit** (Editor)
  - Can edit (but not delete, etc)
  - Implies viewing

### 2.5.4 Roles to Share II

- **Can add** (Contributor)
  - Can add new content below here
  - Does *not* imply viewing or editing
- **Can review** (Reviewer)
  - Can publish/reject
  - Does *not* imply viewing or editing
    - \* Can edit while pending (“clean up”)

New roles do not appear on the sharing tab in Plone 3 (unlike earlier versions of Plone). Adding them requires either editing the Plone source tree, or making new ZCML and code in your product. This is controlled in `lib/python/plone/app/workflow`; see the commented-out code in `localroles.py` and `configure.zcml` for examples of adding the “missing” roles of Owner, Manager, and Member.

As “Editor” doesn’t normally allow deletion or sharing, you can’t make someone a true “collaborator” without sharing the Owner role. You can do this via the ZMI, or by showing the Owner role on the sharing tab.

---

### 2.5.5 Local Role Pattern

```
/projects      <= Share with COO
  /new-office   <= Share with PM
    /goals
    /plans
    /salaries
      /2007.xls <= Share to collaborate
```

- Sharing folder shares *everything* within
  - Including subitems

### 2.5.6 Local Role Blacklisting

```
/projects
  /new-office    <= Shared with Marcus
    /goals
    /plans
  /salaries     <= Should be more secret
    /2007.xls
```

- Can turn off “inherit permissions” for salaries
    - Marcus doesn’t get role here
      - \* Only his “intrinsic” role of Member
- 

### 2.5.7 Logged-in Users Group

- Virtual group for all logged-in users
    - Everyone is member of
    - Useful for assigning local roles to *every* logged in user
      - \* eg, giving Reader role to a private folder, so only logged-in-users can see
- 

### 2.5.8 File Fields Searched

- File fields are now searched
  - If we know how to transform them
    - \* See add-on in ATContentTypes/docs
  - Plone’s File types are searched
  - Add-on Archetypes with FileField fields are searched
- No longer uses TextIndexNG

### 2.5.9 File Fields Search Setup

- Nothing special is needed
    - Changes are all in FileField class
  - Method is memory inefficient for very large files
    - Mark FileField fields as searchable=0 to disable
- 

### 2.5.10 Link Monitoring

- Monitors in-Plone linking
  - Deleting or renaming referent provides warning
  - Can monitor and update in Kupu control panel

## **3 Plone 3 Workflows**

## 3.1 Overview

### Overview

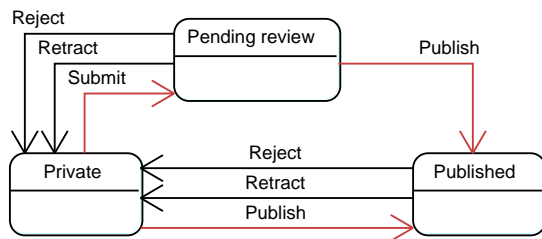
- New Workflows
- Placeful Workflow

## 3.2 New Workflows

---

### 3.2.1 Standard Plone Workflow

- Simple Publication Workflow

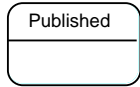


- Owners retract, reviewers reject
- 

### 3.2.2 Simple Publication Workflow

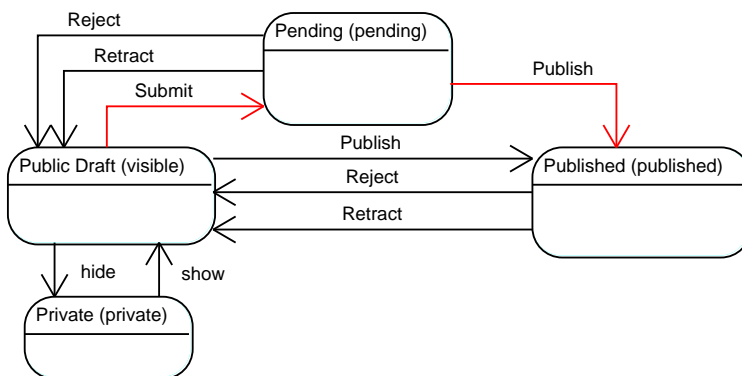
- Things start out as private
  - No more “public draft” (yeah!)
- Owner can edit things when published
  - No more “CNN case”
  - Putting real stuff in workflow transitions

### 3.2.3 One State Workflow



- Single, published state
  - More useful than “no workflow”
    - \* We can control security
    - \* Things looking for “published” will find

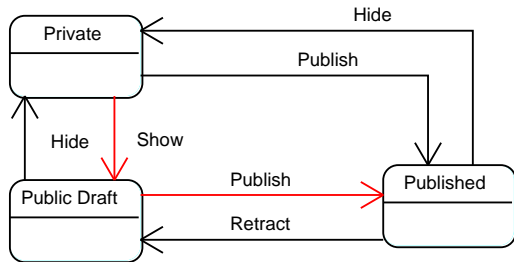
### 3.2.4 Plone Workflow



### 3.2.5 Plone Workflow

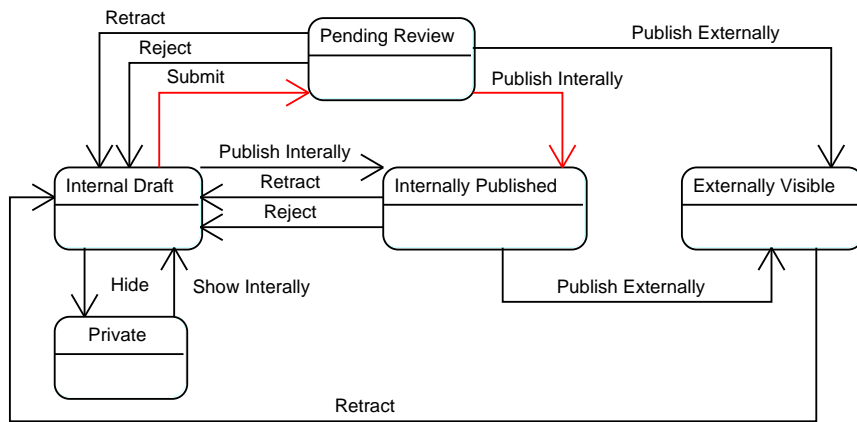
- Almost same as Plone 2 “Plone Workflow”
  - Immediately visible in public draft
  - “CNN case”
    - \* Owner cannot edit while published
- As useless as ever :)

### 3.2.6 Folder Workflow



- Same as 2.5 “Folder Workflow”
- As useless as ever :)

### 3.2.7 Intranet Workflow

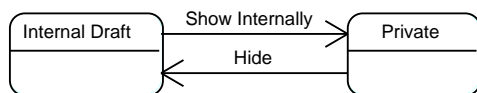


- “Externally visible” is only anon can see

### 3.2.8 Intranet Workflow

- Has “CNN case” (must retract to edit)
  - Initial state is internally-viewable
  - Generally, 2.5 Plone workflow + internal-only stuff
  - Not particularly common case
- 

### 3.2.9 Intranet Folder Workflow



- For folders in Intranet workflow
  - Just private and internal states
- Often better: use Intranet Workflow

## 3.3 Workflow Changes

---

### 3.3.1 Files & Images

- Files & Images don't ship with assigned workflow
    - Fixes common case of forgetting to publish
      - \* Leading to "broken" links/images
  - Can easily assign to a workflow
    - Usually better to assign single-state workflow
- 

### 3.3.2 Default Pages

- Default page workflow is different from folder workflow
  - Constant confusion
- Plone 3 tries to keep in sync
  - When default page undergoes workflow, same transition attempted on folder
    - \* Name must match
  - Transitioning folder does not transition page

## 3.4 Placeful Workflow

---

### 3.4.1 About Placeful Workflow

- Different workflow behavior in different areas of site
  - Useful for “Private” folders
  - New in Plone 2.5; product can be used with 2.1
- 

### 3.4.2 Using Placeful Workflow

- First, create new workflow definitions, if needed
    - eg, private\_plone\_workflow
  - Next, create a new “workflow policy”
    - Site Setup -> Placeful Workflow
- 

### 3.4.3 Establishing Placeful Workflow

- Go to folder on site
- Choose new workflow policy under workflow menu

#### 3.4.4 Advantages of Placeful Workflow

- Can feel “natural” -- uses same content types, etc.
  - But these might have been different content types anyway
  - Useful for changing semantics of entire *tree*
    - \* Harder to do with standard workflow

## 3.5 Customizing Workflows

---

### 3.5.1 Customizing Workflows

- Can still do in ZMI
    - Or in UML
    - Or in Generic Setup XML
  - Often best: design in ZMI, export to GS
- 

### 3.5.2 Transition Action

- **Display in Actions Box URL**
    - In Plone 2, not used
      - \* Couldn't be blank
      - \* Convention was /
- 

### 3.5.3 Transition Action (2)

- **Display in Actions Box URL**
  - In Plone 3, use:
    - `%(content_url)s/content_status_modify?workflow_action=`
    - \* Put transition name at end
  - Could go to specialized form
    - \* To gather more information about transition

## 4 **Versioning**

## 4.1 Overview

### Overview

- Versioning
- Purging
- Staging

## 4.2 Versioning

---

### 4.2.1 Versioning

- CMFEditions is now included in core product
    - As items get edited, each version is saved
    - Can add “Change Note” explaining changes
    - History tab shows history
      - \* Overlap in term “history” with workflow history in byline
- 

### 4.2.2 Features

- Previewing previous versions
  - Reverting to previous versions
  - Comparing (“diff”) to previous versions
- 

### 4.2.3 Versioned Types

- Control of types-to-version on type-by-type basis
  - Controlled in new “types” control panel

#### 4.2.4 Purging

- Normally, all old copies are kept
    - After all packing
    - Increases size of database
      - \* Though minimal impact on performance
  - `portal_purgepolicy` controls this
- 

#### 4.2.5 Staging

- Lightweight, in-site staging
    - Using `iterate` (“Working Copy”)
    - Removes most need for separate sites for staging
  - Included in Plone 3, but not installed
- 

#### 4.2.6 Using Iterate

- “Check out” under content menu
  - Copies content item
    - \* New copy begins at initial workflow state
    - \* Can be edited and workflowed
    - \* Original copy cannot be edited or workflowed

#### 4.2.7 Using Iterate II

- “Check in” under content menu
    - Moves new copy to replace original
      - \* But stays in state of original (ie, homepage stays published)
      - \* History of items are merged
- 

#### 4.2.8 Using Iterate III

- “Cancel checkout” under content menu
    - Delete copies and unlocks original
- 

#### 4.2.9 Iterate Missing Features

- Doesn't stage/version dependencies
    - ie, Images display in content don't change/iterate
    - No branches/tags/etc
      - \* Users would likely find these insanely confusing
- 

#### 4.2.10 Iterate Security

- Two permissions control check-in/out
  - iterate: Check in content
  - iterate: Check out content
- Can add to be controlled in workflow
  - So items must be in certain state to be checked-in, etc.

## 4.3 Plone 3 Practices

### Plone 3 Practices

- CMFEditions is definitely solid
  - Which is good, given that it's now installed by default :)
  - Works great under 2.5!
- iterate is much newer
  - Has Plone 3 dependencies except in antique branches

## 5 Plone 3 Portlets

## 5.1 Adding Portlets

---

### 5.1.1 Adding Portlets

- Can add portlets to root of site
    - Manage portlets link at the root
    - Different portlets request different config
- 

### 5.1.2 Type-Specific Portlets

- Can manage using “Types” control panel
    - eg, mission statement portlet appears only on press releases
- 

### 5.1.3 Group-Specific Portlets

- Can manage using User/Group admin in Control Panel
    - eg, policies portlet appears only for members of Staff
- 

### 5.1.4 Blocking Portlets

- Allows you to have different behavior in different places
  - Block: Don’t show what would normally appear
  - Always Show: Show what would appear if nothing were blocked

### 5.1.5 Useful Add-On Portlets

- `plone.portlet.static`
  - Make a simple static “message” portlet
    - \* eg, “office address”
- `plone.portlet.collection`
  - Make a portlet out of any Collection

## 5.2 Dashboard

### Dashboard

- Users can add portlets here
  - Any portlet type can be added
  - Specific to users
    - \* Though could be used via code with groups/teams/etc

## 6 Content Rules

## 6.1 Overview

---

### 6.1.1 Overview

- Powerful, flexible system for performing actions when something happens
    - Sending mail on publish
    - Moving content on submission
    - Logging all edits
- 

### 6.1.2 Adding a Rule

- Title
- Description
- Event (will clarify with conditions)
- Enabled
  - Simple place to turn off w/o hunting down
- Stop Further Rules
  - If this happens, don't look further

**6.1.3 Conditions**

- Content Type
  - File Extension
  - Workflow State
  - User group or role
  - All must apply
- 

**6.1.4 Actions**

- Log to file
- Notify (web status message)
- Copy/Move/Delete
- Transition
- Send Mail

## 6.2 Limitations

### Limitations

- All conditions must apply
  - No arbitrary scripted conditions
  - Could fix by adding a TALES expression condition
- Email sending is limited in variable replacements
  - Could fix by writing new, better action
  - Or can use a traditional workflow script to send email

## 7 Plone 3 Skinning

## 7.1 Overview

---

### 7.1.1 Overview

- Templates can still live in custom folder
    - Or on-disk skin folder
  - Zope3-wired templates are stored elsewhere
    - Also either on-disk or in-ZODB
- 

### 7.1.2 Zope3 Templates

- Portlets
- Viewlets
  - Page components
    - \* Search box, logo, footer, etc.
- Most “new” templates

## 7.2 **Customize**

---

### 7.2.1 **Customize**

- `portal_view_customizations`
  - Lists all templated-based Zope3 views

Does not list ones that requires class, as these must be customized on disk.

- Can examine to find where they're stored
    - \* Useful for on-disk developers
  - Can be very slow to load on Windows
- 

### 7.2.2 **Customize (2)**

- `portal_view_customizations`
  - Can customize TTW
    - \* Similar to "custom folder" for Zope3 templates
    - \* Cannot customize into different "skin paths"
- Some things can only be customized on disk
  - Changes not captured by GS snapshots (!!)

## 8 Plone 3 Archetypes

## 8.1 Overview

### Overview

- Archetypes
- Different permissions
- Skinning

## 8.2 Archetypes

---

### 8.2.1 Archetypes

- Not going anywhere soon :)
    - Though underpinning are changing, slowly
  - Still the commonly-accepted way to create content types
    - UML is still the easiest & often best way
- 

### 8.2.2 Converting UML

- Need very recent copy of ArchGenXML
  - TTW converter at <http://uml3.joelburton.com>
- 

### 8.2.3 Permissions

- For existing products
  - Permission names have moved
  - See newly-generated product for comparison

### 8.2.4 Skinning

- Can output view widget for field
    - As opposed to using accessor directly
  - Will allow inline editing in Plone 3
    - And other advantages of widget
      - \* LinkField might show URL as link
- 

### 8.2.5 base\_view Example

- yourtype\_view.pt:

```
<div metal:define-macro="body">
  <div metal:use-macro="python:
    context.widget('myfield')" />
</div>
```

## 9 GenericSetup

## 9.1 Introduction

### Overview

- About Generic Setup
- Snapshots
- Creating a New Policy
- Duplication Customization Policies

## 9.2 Generic Setup

---

### 9.2.1 About Generic Setup

- New style setup tool for Plone
  - Plone uses it for its own setup
    - Products can use it for installation
    - You can use it to capture ZMI changes
- 

### 9.2.2 Profiles

- Base Profile
    - Initial setup of site
    - Standard Plone is at `CMFPlone/profiles/default`
    - Can switch to different one later
    - Similiar to Plone 2's "Customization Policies"
- 

### 9.2.3 Extension Profiles

- Either overrides or adds to base profile
  - What most products will use

### 9.2.4 Snapshots

- Can capture state of site
    - Tool settings and content
    - **Not everything** captured
      - \* Only standard and updated tools
- 

### 9.2.5 Snapshots (2)

- Stores results as XML files in ZMI
  - Can compare states
    - Useful to see “what’s changed”
  - Creating snapshots seems to hang sometimes
- 

### 9.2.6 Export

- Creates snapshot of just one step
  - Useful for creating a “site configuration” that captures the configurations of many products
    - eg, capture workflow settings
- 

### 9.2.7 Factoring What You Need

- Can capture changes either with export or snapshot
  - Trim set of files down
  - Trim files themselves to “what you need”

### 9.2.8 Sample Properties.xml

- properties.xml:

```
<?xml version="1.0">
<site>
  <property name="title">Your Site</property>
  <property name="description">Your Description</property>
</site>
```

---

### 9.2.9 Sample Workflows.xml

- workflows.xml:

```
<?xml version="1.0"?>
<object name="portal_workflow"
  meta_type="Plone Workflow Tool">
<object name="foo_workflow" meta_type="Workflow"/>
  <bindings>
    <type type_id="Folder">
      <bound-workflow workflow_id="foo_workflow"/>
    </type>
  </bindings>
</object>
```

---

### 9.2.10 Import

- Apply/re-apply profile steps
  - All Extension profiles and snapshots are listed
- Be careful about “dependencies” box
  - Can re-apply existing policies and overwrite your changes

### 9.2.11 Using with Products

- Create policy
    - Convention: default in folder profiles
    - Register with ZCML
  - Code-generators (ArchGenXML and DIYPloneStyle) do for you
- 

### 9.2.12 GS and ZCML

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:genericsetup=
    "http://namespaces.zope.org/genericsetup">

  <genericsetup:registerProfile
    name="myprod"
    title="Myprod Extension"
    directory="profiles/default"
    description="Extension profile myprod."
    provides="Products.GenericSetup.interfaces.EXTENSION"
  />

</configure>
```

### 9.2.13 GS and QuickInstaller

- Plone 3 introduces “GS-aware” Quick Installer
- An explicit `Install.py` takes precedence, though
  - May need to pull in GS manually, then, in `Install.py`:

```
def install(self, reinstall=False):
    id = "ourproduct:default"
    portal_setup = getToolByName(self, 'portal_setup')
    portal_setup.runAllImportStepsFromProfile(
        'profile-%s' % extension_id,
        purge_old=False)
```

---

### 9.2.14 Getting GS w/o ZCML

- In product `init.py`:

```
from Products.GenericSetup import EXTENSION, profile_registry

def initialize(context):
    profile_registry.registerProfile(
        'profile-id',
        'Profile Title',
        'Profile Description',
        'path-to-profile',
        'ProductName',
        EXTENSION,
    )
```

---

### 9.2.15 GS and Python Code

- Can have generic setup run any Python code

### 9.2.16 GS and Python Code II

- add to `import_steps.xml`:

```
<import-step id="foopolicy"
  version="1.0"
  handler="Products.Foo.myPolicy.myPolicy"
  title="My Policies">
  My special things
</import-step>
```

---

### 9.2.17 GS and Python Code III

- `myPolicy.py`:

```
def mypolicy(context):
    site = context.getSite()
    site.invokeFactory(
        'Document',
        'foo',
        title='Foo Document')
```

## 9.3 Exercises

### 9.3.1 Exercises

1. Create a simple product with just a GenericSetup profile. In the profile, clone the News Item content type into a Press Release type.
- 

### 9.3.2 Exercises Answers #1

1. Create a simple product...

- Create GSPlay folder in Products

- `__init__.py`:

```
# Make this into a package
```

- `configure.zcml`:

```
<configure
  xmlns="http://namespaces.zope.org/zope">

  <include file="profiles.zcml" />

</configure>
```

**9.3.3 Exercises Answers #2**

- profiles.zcml:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:genericsetup=
    "http://namespaces.zope.org/genericsetup">

  <genericsetup:registerProfile
    name="default"
    title="GSPlay Extension"
    directory="profiles/default"
    description="Extension profile for GSPlay."
    provides="Products.GenericSetup.interfaces.EXTENSION"
  />

</configure>
```

---

**9.3.4 Exercises Answers #3**

- Export “Types Tool” step from portal\_setup
- Trim to profiles/default/types.xml:

```
<?xml version="1.0"?>
<object name="portal_types" meta_type="Plone Types Tool">
  <object name="PressRelease"
    meta_type="Factory-based Type Information with dynamic views"/>
</object>
```

---

**9.3.5 Exercises Answers #4**

- Delete all types from types/ except News\_Item.xml
- Rename to PressRelease.xml
- Change properties (title, description, etc.)



## 10 Buildout Basics

## 10.1 Overview

### Overview

- What is buildout?
  - Who is it for?
- Basic usage

## 10.2 Buildout

---

### 10.2.1 Buildout

- System for installing and configuring Plone
    - And add-on products
    - And, possibly, Zope and Python
    - In a scripted, reproducible way
- 

### 10.2.2 Who Is It For?

- Developers
  - Experienced, advanced integrators
    - Multiple rollouts of same setup
    - Complex interdependencies of products
    - Desire to get lost in poor docs and bleeding edge :)
- 

### 10.2.3 Idea

- A single config file, `buildout.cfg`
  - This downloads/generates/configs all components
  - You don't/can't touch most things below it
- Very reproducible--just copy `buildout.cfg` to other box

### 10.2.4 Getting Started

- Martin's book
- 

### 10.2.5 Eggs

- Pure Python idea, not a Zope/Plone thing
    - A package with installation scripts & metadata
    - Can be a directory
    - Can be a zipfile of a directory
      - \* And can be used without unzipping
- 

### 10.2.6 Eggs 2

- Many Plone products distributed as “eggs”
    - Can still unzip code in `lib/python`, as normal
    - Or run their `setup.py`
  - Most Plone products in eggs can't be used as zip file
    - But this will change
- 

### 10.2.7 Challenges

- *Very* command-line oriented
  - Complex on Windows
- Option location & spelling is different than in traditional `zope.conf`, `zeo.conf`, etc.
- Errors can be opaque

### 10.2.8 But ...

- The developers love it!
    - So it will get better
  - At the point you need it, you can learn the details
- 

### 10.2.9 Does this Replace My Installer?

- No. Yes. Maybe.
    - Installers are much easier!
      - \* Easy is good
    - More likely, installers will be a “front end” for buildout
      - \* Start easy, can edit `buildout.cfg` when you need to
- 

### 10.2.10 Eggs without Buildout

- Some new-style products (unzipped):

```
plone.portlets.joel/  
  __init__.py  
plone/  
  __init__.py  
  portlets/  
    __init__.py  
    joel/  
      __init__.py  
      foo.py  
      bar.py  
      browser/
```

### 10.2.11 Eggs without Buildout 2

- Can't just unzip this, as there already *is* a plone package
    - And may even be a portlets package inside of it!
  - The goal is you can put this in top-level, next to `plone.portlets.bob`, and they'd both work
  - The `__init__.py` files in top-level, in `plone/`, and in `portlets/` contain magic name-mangling stuff
- 

### 10.2.12 Eggs without Buildout 3

- Copy `joel/` folder and put in existing `plone/portlets` dir
  - If you don't have `plone`, but not `plone/portlets`:
    - \* Copy `portlets/` into `plone`
    - \* **Empty** the `portlets/__init__.py` file but keep the file
      - That removes the weird name-mangling stuff

## 11 Plone 3 and Python

## 11.1 Overview

### Overview

- Classes
- Decorators
- Unicode strings

## 11.2 Classes

---

### 11.2.1 Classes

- No changes, but more need
    - Portlets are classes
    - “Views” (templates) can use classes
    - Utilities and adapters are classes
    - ... and more!
- 

### 11.2.2 Classes 2

- Nothing new. Just review!

```
class Dog(Animal):
    """Our canine friend"""

    color = "spotted"

    def __init__(self, color, name):
        ...

    def rescue(self, severity=5):
        """Go, Lassie, Go!"""
        ...
```

---

### 11.2.3 Classes 3

- Make sure you understand `init` and `super`
  - Python in a Nutshell has excellent chapter on OO

### 11.2.4 Decorators

- A “decorated function” is a function with a wrapper around it
    - Can be used for interesting meta-programming
- 

### 11.2.5 Decorator Example

- eg, every time a function is called, log it:

```
@somelogger
def myMethod(a, b, c)
    ...
```

- Exact same thing as:

```
def myMethod(a, b, c):
    ...
myMethod = somelogger(myMethod)
```

---

### 11.2.6 Decorators

- Used in z3-technology to “mark” functions/methods
  - “This is an adapter”
  - “This is a view”
  - “This is protected in some way”

### 11.2.7 Unicode

- Unicode = strings can be more than 8-bit
  - Required for many non-English characters
- "Dog" is string; u"Dog" is unicode
  - Also 'Cat'/u'Cat' and """"Long"""/u""""Long"""
  - They're **not** the same
  - z3 stuff fail w/strings if expecting unicode

## 12 Interfaces and ZCML

## 12.1 Overview

### Overview

- Interfaces
- Marker interfaces
- ZCML

## 12.2 What Is Zope 3

---

### 12.2.1 Benefits of Zope 3

- “Pure Python” feel
  - More declarative; less “magical”
  - Comprehensive suite of tests
  - Good for writing applications that have “components”
- 

### 12.2.2 Downsides of Zope 3

- Another technology to learn!
  - And not one aiming for beginners
- No TTW configuration for many things

## 12.3 Interfaces

---

### 12.3.1 Interfaces

- The “public” face of a class
    - Explains what the class does
    - But not how it does it
  - Useful for documentation
    - And for interoperability with other programmers
- 

### 12.3.2 Student

```
class Student:
    """Student of a bootcamp."""

    def _isBirthday(self):
        # ... code goes here ...

    def isClassFree(self):
        """Does this student get a free class?

        They get a free class on their birthday.
        """
        # ... code goes here ...
```

### 12.3.3 Student Interface

- interfaces.py:

```
from zope.interface import Interface

class IStudent(Interface):
    """Student of bootcamp."""

    def isClassFree():
        """Does this student get a free class?"""
```

---

### 12.3.4 Student Implementation

- student.py:

```
from zope.interface import implements

class Student:
    implements(IStudent)

    def _isBirthday(self):
        # ... code goes here ...

    def isClassFree(self):
        # Give them a class on their birthday
        # ... code goes here ...
```

### 12.3.5 Advantages of Interfaces

- Obvious how interact with object
    - What is public vs. not
  - Don't have to page through code to read docstrings
  - The interface can be used by others
    - Who want to implement a *different* kind of student
    - But still have the same API
- 

### 12.3.6 Disadvantages of Interfaces

- More typing
- Two places to read
- May be overkill for simple, obvious objects with small teams

## 12.4 Marker Interfaces

---

### 12.4.1 “Marker Interfaces”

- An interface that “means” something
    - But doesn’t suggest any implementation
- 

### 12.4.2 Zope 2-Style Markers

- student.py:

```
class Student:
    _canAppearInBlog = True

    ...
```
  - Z2 uses attributes to “signal” various things
    - eg `hasattr(self, '_canAppearInBlog')`
- 

### 12.4.3 Zope 3-Style Markers

- interfaces.py:

```
from zope.interface import Interface

class IBloggable(Interface):
    """Object can appear in blog."""
```

#### 12.4.4 Zope 3-Style Markers II

- student.py:

```
from interfaces import IBloggable
from zope.interfaces import implements

class Student:
    implements(IBloggable)
```

## 12.5 ZCML

---

### 12.5.1 ZCML Intro

- ZCML = XML dialect for Zope3 configuration
    - Mark class as implementing interfaces
    - Can be stored in top-level `configure.zcml` file
- 

### 12.5.2 Example ZCML

- `configure.zcml`:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
>
  ...
</configure>
```

---

### 12.5.3 Example ZCML: Five

- May require `xmlns` attributes for extensions (`five`, `browser`, etc)
- `configure.zcml`:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:five="http://namespaces.zope.org/five"
>

</configure>
```

### 12.5.4 Marking Class with Iface

```
<five:implements  
    interface=".interfaces.IBloggable"  
    class=".content.student.Student" />
```

---

### 12.5.5 Finding Things in ZCML

- "." refers to current package:
    - `.interfaces.IBloggable`
  - Or give full Python package:
    - `PIL.ImageImage`
  - Or look in Zope/Plone products:
    - `Products.CMFPlone.interfaces`
- 

### 12.5.6 Marker Interface TTW

- "Interfaces" tab lists interfaces
  - Can mark on for a class
- Nothing critical there yet
  - But addon products may add
  - And so might future Plones

## 12.6 Plone 3 Practices

---

### 12.6.1 Hiding From Breadcrumbs

- One object?
    - Can mark in ZMI with interfaces tab
  - Your code?
    - Can add to class:

```
class Student:  
    implements(IHideFromBreadcrumbs)
```
- 

### 12.6.2 Hiding From Breadcrumbs II

- General, best practice
  - Mark class in ZCML
  - Doesn't mix code and configuration

## 12.7 ZCML Organization

---

### 12.7.1 ZCML Organization

- Can split up ZCML into multiple files
  - Only `configure.zcml` is examined
  - It can include other files:

```
<include file="..." [package="..."] />
```

---

### 12.7.2 ZCML Organization II

- `configure.zcml`:

```
<include file="implements.zcml" />
```
  - Note this is a full filepath, not Py package
    - eg, no leading period
- 

### 12.7.3 ZCML Organization III

- `configure.zcml`:

```
<include package=".browser" />
```

  - Reads `configure.zcml` from `browser/`

## 12.8 Exercises

### 12.8.1 Exercises

1. Hide all normal Plone Folders from the breadcrumbs/navigation.
    - These are things that all of class `ATFolder`, found in `ATContentTypes`
    - The interface for hide-from-breadcrumbs is in `CMFPlone.interfaces.breadcrumbs`
- 

### 12.8.2 Exercises Answers #1

1. Hide all normal Plone Folders from the breadcrumbs/navigation.

- `configure.zcml`:

```
<five:implements
  class="Products.ATContentTypes.content.folder.ATFolder"
  interface="Products.CMFPlone.interfaces.breadcrumbs.IHideFromBreadcrumbs"
/>
```

## 13 Utilities and Adapters

## 13.1 Overview

### Overview

- Utilities
- Named Utilities
- ZCML Overriding
- Adapters

## 13.2 Utilities

---

### 13.2.1 Utilities

- Single tools
    - Similar to in-ZODB CMF tools
    - Can be replaced/extended in other products
- 

### 13.2.2 Making a Utility

- Define an interface for it:

```
class ICapitalizer(Interface):
    """Capitalize phrases."""

    def __call__(msg):
        """Return the capitalized phrase."""
```

---

### 13.2.3 Making a Utility II

- Create a utility class:

```
class SimpleCapitalier(object):
    implements(ICapitalizer)

    def __call__(self, msg):
        return msg.upper()
```

### 13.2.4 Making a Utility III

- Register the utility (in ZCML):

```
<utility factory=".capitalizer.SimpleCapitalizer" />
```

- You can also add a `provides` attribute for which interface it implements  
This is only needed if more than one is possible, or if none are specified in class.
- 

### 13.2.5 Using Utilities

- Get the current implementation:

```
from zope.component import getUtility  
cap = getUtility(ICapitalizer)  
cap("this is lower")
```

## 13.3 Named Utilities

---

### 13.3.1 Named Utilities

- A utility that has a name
    - Can have more than one with different names
    - Can look up by ( name + interface)
  - Often used for “registries”
    - Transformations, conversions, etc.
- 

### 13.3.2 Making a Named Utility

- Still make an interface and class
- Register:

```
<utility
  factory=".capitalizer.SimpleCapitalizer"
  name="english"
/>
```

---

### 13.3.3 Using a Named Utility

```
from zope.component import getUtility
cap = getUtility(ICapitalizer, name="english")
cap("this is lower")
```

## 13.4 **Overriding ZCML**

---

### 13.4.1 **Conflicting ZCML**

- ZCML declarations cannot conflict
    - eg, registering two classes as same non-named utility
  - An error during startup will occur
- 

### 13.4.2 **Overriding ZCML**

- To override ZCML, put in `overrides.zcml` file in your product
  - This file is looked for automatically
  - It overrides the normal layer of `configure.zcml`

## 13.5 Products Versus Packages

---

### 13.5.1 Products? Packages?

- Zope 2 “products” are Python packages
    - But not in normal PYTHONPATH
      - \* Magic required to find
  - Zope 3-style pure packages are in PYTHONPATH
    - A normal package *might* be useful outside of Zope/Plone
      - \* But most still have lots of Zope/Plone deps
- 

### 13.5.2 Products? Packages? II

- Products are scanned automatically
    - For `configure.zcml`
    - And `__init__.py`
  - Packages are not
    - Must add a “slug” to notice `configure.zcml`
- 

### 13.5.3 ZCML Slugs

- A `.zcml` file, typically just including `zcml` for a package
  - Stored in `etc/package-includes`
- Conventionally, named `[product]-configure.zcml`
  - Or `[product]-overrides/zcml`

#### 13.5.4 Sample ZCML Slug

- `myproduct-configure.zcml`:

```
<include package="myproduct" />
```

## 13.6 Adapters

---

### 13.6.1 Adapters

- Method of “adding methods” to classes
    - Similar to subclassing, but more flexible
- 

### 13.6.2 Subclassing

- zoo.py

```
class Mammal:
    ...

class Dog(Mammal):
    ...

class Person(Mammal):
    ...
```
- 

### 13.6.3 Subclassing Problems

- Difficult to manage in large systems
  - Typical Plone object has 15+ subclasses!
- Adding new functionality is tricky
  - Since existing objects don’t include that subclass

### 13.6.4 Adapter Benefit

- New object “adapts to” interface
    - Creates a temporary object with additional, discovered functionality
- 

### 13.6.5 Adapter Example

- `interfaces.py`:

```
class INoise(Interface):
    def makeNoise():
        """Make right kind of noise for this animal."""
```

---

### 13.6.6 Adapter Example

- `interfaces.py`:

```
class IDog(Interface):
    """Canines"""
```

- `dog.py`:

```
class Dog:
    implements(IDog)
```

---

### 13.6.7 Adapter Example

- `bark.py`:

```
class Bark:
    implements(INoise)

    def makeNoise(self): return "woof!"
```

### 13.6.8 Adapter Example

- Once we do the “wiring”, we can “adapt” bark.py to dog.py

```
>>> from interface import INoise
>>> mydog = Dog()
>>> INoise(mydog).makeNoise()
```

- This finds the “right” INoise adaption for Dogs
    - Since Dogs implement the IDog interface
- 

### 13.6.9 Example ZCML

- Adapt bark as INoise for dog:

```
<adapter
  for=".interfaces.IDog"
  provides=".interfaces.INoise"
  factory=".bark.Bark" />
```

---

### 13.6.10 Plone 3 Techniques

- Adapters used to add logic onto classes
  - Might not have to create separate content types
  - Experimental products allow new AT fields
- One day, all content types may be one class

## 13.7 Exercises

### 13.7.1 Exercises

1. Create a package, `languagetools`, that contains and registers the capitalization utility from this chapter.
    - Register it as a global, unnamed utility
    - Write an external method (or unit test) that finds this utility and tests it.
- 

### 13.7.2 Exercises Answers #1

1. Create a product, `languagetools`, that contains and registers the capitalization utility from this chapter.
    - Create `languagetools` in `lib/python`
    - Add an empty `__init__.py` to it
    - Create the slug in `etc/package-includes/languagetools-configure.zcml`:
 

```
<include package="languagetools" />
```
- 

### 13.7.3 Exercise Answers #2

1. Create a product, `languagetools` ...
  - Add `interfaces.zcml`:

```
class ICapitalizer(Interface):
    """Capitalize phrases."""

    def __call__(msg):
        """Return the capitalized phrase."""
```

### 13.7.4 Exercise Answers #3

1. Create a product, languagetools ...

- Add `capitalizer.zcml`:

```
class SimpleCapitalier(object):
    implements(ICapitalizer)

    def __call__(self, msg):
        return msg.upper()
```

---

### 13.7.5 Exercise Answers #4

1. Create a product, languagetools ...

- `configure.zcml`:

```
<configure
    xmlns="http://zope.namespaces.org/zope">
    <utility factory=".capitalizer.SimpleCapitalizer" />
</configure>
```

---

### 13.7.6 Exercise Answers #4

1. Create a product, languagetools ...

- `INSTANCE_HOME/Extensions/testUtility`:

```
from zope.component import getUtility

def testUtility(self):
    """Test utility"""
    cap = getUtility(ICapitalizer)
    return cap("this is lower")
```

- Register external method with `module='testUtility'` and `function='testUtility'`

## 14 Views and View Classes

## 14.1 Overview

### Overview

- “Views”
- View Classes

## 14.2 Views

---

### 14.2.1 Views

- “View” is a horribly overloaded word in Zope/Plone
    - These are “on-disk, ZCML-wired Page Templates”
    - Get to choose where they can be used
      - \* Rather than a “global namespace”
- 

### 14.2.2 Sample View

- `configure.zcml`:

```
<browser:page
  for="*"
  name="mypage.html"
  template="mytemplate.pt"
  permission="zope2.View"
/>
```

- Traversing to `mypage.html` on *anything* shows `mytemplate.pt`

### 14.2.3 Sample View II

- Requires “browser” namespace:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser"
  >
  <browser:page
    for="*"
    name="mypage.html"
    template="mytemplate.pt"
    permission="zope2.View"
  />
</configure>
```

---

### 14.2.4 Sample View III

- Not much advantage so far
    - Could be in CMF on-disk skin folder
    - But can tie to certain interfaces
- 

### 14.2.5 Sample View IV

```
<browser:page
  for=".interface.IMyNewType"
  name="mypage.html"
  template="mytemplate.pt"
  permission="zope2.View"
/>
```

- Now, only works for things implementing IMyNewType
  - Doesn't “pollute” global namespace

## 14.3 Permissions

---

### 14.3.1 Permissions

- Same permissions are being used
    - But with new, “package-style” names
    - View => zope2.View
      - \* Convention: Remove spaces, jam together
      - \* Prefix with product (cmf.ModifyPortalContent)
        - See common in Products/Five/permissions.zcml
- 

### 14.3.2 New Permissions

- Can add permissions as:

```
<permissionid="zope2.View"  
  title="View"  
>
```

---

### 14.3.3 Significant Security Change

- ZCML wired PageTemplate views are *trusted*
  - They can do things currently logged-in users cannot
  - **Very significant** security change
    - \* You must trust people who write these!

#### 14.3.4 The Goggles

- In Zope3 + Plone3, often see URLs like:

```
http://site/@mypage.html
```

- @@ tells Zope it's a view
    - Not required, /mypage.html would work, too
    - But disambiguates if "mypage.html" were a content object in ZODB
      - \* Or method / something else acquired
- 

#### 14.3.5 View Classes

- Provides a Python class that can be associated with PageTemplates
    - Can be used to handle complex logic, etc.
- 

#### 14.3.6 Zope2-style "View Class"

- area.py PythonScript:

```
##parameters=room
from math import pi
radius = room.getRadius()
return pi * radius * radius
```

---

#### 14.3.7 Zope2-style "View Class"

- room\_info.pt:

```
<span tal:replace="python: context.area(context)">
  [room area]
</span>
```

### 14.3.8 Zope2-style Overview

- Easy to do & understand
  - Loose wiring (“coupling”) between template + code
  - Not easy to change area calculation for one type of content
- 

### 14.3.9 Zope3-style View Class

- area.py:

```
from math import pi
from Products.Five import BrowserView

class RoomView(BrowserView):
    def getArea(self):
        radius = self.context.getRadius()
        return pi * radius * radius
```

---

### 14.3.10 Zope3-style View Class

- The ZCML:

```
<browser:page
    for=".interfaces.IRoom"
    name="area"
    class=".area.RoomView"
    permission="zope2.View"
/>
```

### 14.3.11 Using Zope3 View Class

- room\_info.pt:

```
<span tal:define="viewc context/@@area"
      tal:replace="python: viewc.getArea(context)">
  [room area]
</span>
```

- @@ are optional, but clarifying
- 

### 14.3.12 Binding Class and View Together

- In ZCML, can connect the template and the class:

```
<browser:page
  for=".interfaces.IRoom"
  name="room_info"
  class=".area.RoomView"
  template="room_info.pt"
  permission="zope2.View"
/>
```

- In template, just refer to view/getArea
- 

### 14.3.13 Binding Class and View Together II

- Simplifying
  - Requires page template being ZCML-wired

## 14.4 Plone 3 Practices

---

### 14.4.1 “Browser” Package

- Views and view classes often go into browser/ package of product
    - Browser-specific ZCML into browser/configure.zcml
    - Don’t forget to include this in main configure.zcml
    - Don’t forget to make a package (`__init__.py`)
- 

### 14.4.2 “Templates” Directory

- Where there are more than a few page templates, these go in templates directory of browser package

## 15 Viewlets

## 15.1 Overview

### Overview

- Viewlets
- Re-arranging and hiding
- Viewlet managers
- Layers

---

## 15.2 Viewlets

---

### 15.2.1 Viewlets

- Small, self-contained visual item
    - Search box
    - Logo
  - Portlets are a special kind of viewlet
- 

### 15.2.2 Re-Arranging Viewlets

- `/manage-viewlets`
    - Re-arrange within manager
    - Hide
  - Changes persist in ZMI
    - Can be captured by GS snapshot
- 

### 15.2.3 A Viewlet

- `sponsor.pt`:

```
<p>
  Plone is brought to you by the letter P
  and the numbers 4, 5, and 6.
</p>
```
- Nothing very special about it :)

### 15.2.4 Viewlet ZCML Registration

- `configure.zcml`:

```
<browser:viewlet
  name="yourproduct.sponsorship"
  for="*"
  manager=
"plone.app.layout.viewlets.interfaces.IPortalFooter"
  template="sponsor.pt"
  permission="zope.Public"
/>
```

---

### 15.2.5 A Viewlet with Logic

- `webmaster.pt`:

```
<p>Our webmaster is at
  <i tal:replace="view/webmaster_email">[email]</i>
</p>
```

---

### 15.2.6 Viewlet Class

- `webmaster.py`:

```
class WebmasterViewlet(ViewletBase):

    render = ViewPageTemplateFile('webmaster.pt')

    def update(self):
        super(WebmasterViewlet, self).update(self)
        portal = self.portal_state.portal()
        self.webmaster_email = portal.email_from_address
```

### 15.2.7 Viewlet ZCML Registration

- `configure.zcml`:

```
<browser:viewlet
  name="yourproduct.webmaster"
  for="*"
  manager=
"plone.app.layout.viewlets.interfaces.IPortalFooter"
  class=".webmaster.WebmasterViewlet"
  permission="zope.Public"
/>
```

## 15.3 Re-Arranging and Hiding

---

### 15.3.1 Rearranging and Hiding

- Done in GS profile, not in ZCML
    - Weird mix of GS and ZCML in viewlets
- 

### 15.3.2 Specifying Viewlet Order

- profiles/default/viewlets.xml:

```
<object>
  <order manager="plone.portalfooter"
        skinname="YourSkinPath">
    <viewlet name="webmaster.sponsorship" />
    <viewlet name="webmaster.webmaster" />
  </order>
</object>
```

---

### 15.3.3 Removing Viewlets

- Leaving out of viewlets.xml doesn't hide
  - They just appear at the bottom of that manager
- Hide using a `<hidden />` directive

### 15.3.4 Hiding Viewlets

- profiles/default/viewlets.xml:

```
<object>
  <hidden manager="plone.portalfooter"
    skinname="YourSkinPath">
    <viewlet name="plone.footer" />
    <viewlet name="plone.colophon" />
  </order>
</object>
```

---

### 15.3.5 Inserting in Order

- profiles/default/viewlets.xml:

```
<object>
  <order manager="plone.portalfooter"
    skinname="YourSkinPath"
    based-on="Plone Default">
    <viewlet name="myproduct.webmaster"
      insert-after="plone.footer" />
  </order>
</object>
```

- Can also insert-before
- 

### 15.3.6 Viewlet Managers

- Container for viewlets
  - Plone ships with several (plone.portaltop, plone.portalfooter, etc)
  - Can create your own
    - \* They're just an interface

### 15.3.7 New Viewlet Manager

- `interfaces.py`:

```
from zope.viewlet.interfaces import IViewletManager

class IPortalHelp(IViewletManager):
    """Viewlet manager for help stuff at bottom."""
```

---

### 15.3.8 Adding Viewlets to Manager

- Same as before, but with new manager line.
- `configure.zcml`:

```
<browser:viewlet
  name="webmaster.sponsorship"
  for="*"
  manager=".interfaces.IPortalHelp"
  class=".webmaster.WebmasterViewlet"
  permission="zope.Public"
/>
```

## 15.4 Layers

---

### 15.4.1 Layers

- Analogous to CMF “skin paths”
    - New skin products can create one
      - \* `DIYPloneStyle` does this for you
  - To ensure an item only appears in your theme
- 

### 15.4.2 Making a Layer

- Normally, done for you by `DIYPloneStyle`:

```
<interface
  interface=".interfaces.IThemeSpecific"
  type="zope.publisher.interfaces.browser.IBrowserSkinType"
  name="mytheme"
/>
```

---

### 15.4.3 Binding Viewlets to Theme

- `configure.zcml`:

```
<browser:viewlet
  name="webmaster.sponsorship"
  for="*"
  manager=".interfaces.IPortalHelp"
  class=".webmaster.WebmasterViewlet"
  permission="zope.Public"
  layer=".interfaces.IThemeSpecific"
/>
```

- Note we bind to *interface*, not *name*!



## **16 Plone 3 Portlets**

## 16.1 Overview

### Overview

- Classic Portlets
- New-Style Portlets

## 16.2 Classic Portlets

---

### 16.2.1 Classic Portlets

- Straightforward, template-based
    - Much easier to write
    - Slightly faster
    - Use use scripts/view classes for logic
    - Can use dynamic placement of new system
    - Can't have "configuration-while-adding" of new portlets
- 

### 16.2.2 Sample Classic Portlet

- portlet\_mission.pt:

```
<dl class="portlet" metal:define-macro="portlet">
  <dt class="portletHeader">Mission Statement</dt>
  <dd class="portletItem even">Our first goal...</dd>
  <dd class="portletItem odd">Our second goal...</dd>
  <dd class="portletFooter">More...</dd>
</dl>
```

- Register with template portlet\_mission and macro portlet

## 16.3 New-Style Portlets

---

### 16.3.1 New-Style Portlet

- Three Python classes
    - Interface of portlet data requirements
    - Assignment
      - \* Controls where/when it can appear
    - Renderer
      - \* Handles how it appears
- 

### 16.3.2 New Mission Portlet

- `mission.py`:

```
from plone.portlets.interfaces import IPortletDataProvider

class IMissionPortlet(IPortletDataProvider):

    statement = schema.Text(title=u"Mission Statement",
                           description=u"Enter your mission statement",
                           required=True)
```

- Schema here is Zope 3's `formlib`

### 16.3.3 New Mission Portlet II

- mission.py (cont'd):

```
from plone.app.portlets.portlets import base

class Assignment(base.Assignment):
    implements(IMissionPortlet)

    title = u"Mission Statement"

    def __init__(self, statement):
        self.statement = statement
```

---

### 16.3.4 New Mission Portlet III

- mission.py (cont'd):

```
class Renderer(base.Renderer):
    _template = ViewPageTemplateFile('mission.pt')

    def render(self):
        return self._template()
    def statement(self):
        return self.data.statement
    def title(self):
        return self.data.title
```

- Most similar to “view class” for portlet

### 16.3.5 New Mission Portlet IV

- Need mostly-boilerplate add/edit classes
- `mission.py` (cont'd):

```
from plone.app.portlets.portlets import base

class AddForm(base.AddForm):
    form_fields = form.Fields(IMissionPortlet)
    label = u"Add Mission Portlet"
    description = u"Displays mission statements."

    def create(self, data):
        return Assignment(statement=data.get('statement', ''))
```

---

### 16.3.6 New Mission Portlet V

- (cont'd):

```
class EditForm(base.EditForm):
    form_fields = form.Fields(IMissionPortlet)
    label = u"Edit Mission Portlet"
    description = u"Displays mission statements."
```

### 16.3.7 New Mission Portlet VI

- Need ZCML registration in `configure.zcml`:

```
<configure xmlns:plone="http://namespaces.plone.org/plone">
  <plone:portlet
    name="portlets.Mission"
    interface=".mission.IMissionPortlet"
    assignment=".mission.Assignment"
    renderer=".mission.Renderer"
    addview=".mission.AddForm"
    editview=".mission.EditForm"
  />
</configure>
```

---

### 16.3.8 New Mission Portlet VII

- `portlet.pt`:

```
<dl class="portlet">
  <dt class="portletHeader"
    tal:content="view/title">[title]</dt>
  <dd class="portletItem"
    tal:content="view/statement">[statement]</dd>
</dl>
```

- Doesn't require any `<metal>` statements
- Notice use of `renderer` as view class

### 16.3.9 New Mission Portlet VIII

- Requires GS install, profiles/default/portlets.xml:

```
<?xml version="1.0"?>
<portlets>
  <portlet
    addview="portlets.Mission"
    title="Mission Portlet"
    description="A portlet showing mission."
  />
</portlets>
```

---

### 16.3.10 Classic or New Style

- Clearly, new style have more conceptual overhead/typing
    - But lots of flexibility in assignment class
      - \* In cases of complex logic in placement
      - \* Keeps “should-it-show” logic out of template itself
- 

### 16.3.11 Classic or New Style II

- Alternate strategy for our mission portlet
  - Sitewide tool where mission statement is kept, shown in classic portlet
    - \* Using AGX’s <<tool>> stereotype

### 16.3.12 **Classic or New Style III**

- Best strategy in many cases:
  - Use `plone.portlet.static` and `plone.portlet.collection`
  - New-style portlets for simple messages or Collection results

## 17 Formlib Intro

## 17.1 Overview

---

### 17.1.1 Overview

- Zope3 form generation
    - Can be used to make freestanding forms
      - \* Though PloneFormGen is often much nicer
    - Used for modeling data required by portlets, etc.
- 

### 17.1.2 Concepts

- An interface defines the data requirements
- An implementation class defines how it works

## 17.2 Example

---

### 17.2.1 Example 1

- myform.py:

```
from zope.interface import Interface
from zope import schema

class IMyForm(Interface):
    """Define our fields"""

    name = schema.TextLine(title=u"Name",
                           required=True)
    subject = schema.Text(title=u"Subject")
```

---

### 17.2.2 Implementation Class

- Uses fields defined in interface
- Has action methods
  - Decorator makes them into buttons

### 17.2.3 Example 2

- `myform.py`:

```
from zope.formlib import form
from Products.Five.formlib import formbase

class MyForm(formbase.PageForm):
    form_fields = form.FormFields(IMyForm)
    label = u"Fill out My Form!"

    @form.action(u"Send")
    def action_send(self, action, data):
        """Do something with data"""
        raise "You said", data
```

---

### 17.2.4 Registering with ZCML

- Gets registered as a normal view
    - The “form” magic comes from the base class
- 

### 17.2.5 Example 3

- `configure.zcml`:

```
<browser:page
    for="*"
    name="myform"
    class=".myform.MyForm"
    permission="zope2.View"
/>
```

### 17.2.6 Common Field Types: String

- ITextLine
    - Like AT's "String"; no newlines
  - IText
    - Like AT's "Text"; allows newlines
  - IASCIILine, IASCII
    - Only low-bit ASCII; otherwise, see above
- 

### 17.2.7 Common Field Types: Numbers

- IInt
  - IFloat
- 

### 17.2.8 Common Field Types: Dates

- IDatetime
- IDate

### 17.2.9 Common Field Types: Other

- IBool
    - Yes/No choice
  - IURI
    - Absolute URL
  - ICollection
    - Like “lines” field
    - Unrelated to “Plone 3 collections” (ie, Smart Folders)
- 

### 17.2.10 Common Field Parameters

- title
- description
- required
- readonly
- default

## **17.3 Road Ahead**

### **Road Ahead**

- Excellent examples on how to use in Philip's book

## **18 KSS Intro**

## 18.1 Overview

---

### 18.1.1 Overview

- KSS Overview
  - Simple Client-Side Actions
  - Server-Side Actions
- 

### 18.1.2 What is KSS?

- “Kinetic Style Sheets”
  - Power of JS, syntax of CSS
  - Allows you to declare behavior
  - Includes AJAX library
- Very powerful, very cool!

## 18.2 Client-Side

---

### 18.2.1 Calculator Form

- calc.pt:

```
<html metal:use-macro="context/main_template/macros/master">
<div metal:fill-slot="main">

<form>
  2 x <input type="text" id="numfield" /> =
  <b id="result">?</b>
</form>

</div>
</html>
```

- Nothing unusual or KSS-specific
- 

### 18.2.2 KSS

- calc.kss:

```
#result:click {
  action-client: alert;
  alert-message: "Hello";
}
```

### 18.2.3 KSS Syntax

- calc.kss:

```
#result:click {  CSS identifier + event
  action-client: alert;  Action
  alert-message: "Hello"; Parameters
}
```

---

### 18.2.4 KSS 2

- calc.kss:

```
#result:click {
  action-client: alert;
  alert-message: formVar('numfield');
}
```

- formVar is a “parameter provider” (helper function)

## 18.3 Server-Side

---

### 18.3.1 KSS

- play.kss:

```
#num:change {  
  action-server: kssCalc;  
  kssCalc-num: currentFormVar('numfield');  
}
```

- Calls script “kssCalc”
    - Can be view, PythonScript, ExternalMethod, whatever
    - Passes form field numfield as num
- 

### 18.3.2 Script

- kssCalc.py (PythonScript):

```
request = context.REQUEST  
answer = str(2 * int(num))          # get answer  
  
from kss.core.ttwapi import ( startKSSCommands,  
                              getKSSCommandSet, renderKSSCommands )  
  
startKSSCommands(context, request) # boilerplate  
  
core = getKSSCommandSet('core')  
core.replaceInnerHTML('#result', answer)  
  
return renderKSSCommands()         # boilerplate
```

### 18.3.3 Alternate Script

- As on-disk class, `calculator.py`:

```
from kss.core import kssaction
from plone.app.plonekssview import PloneKSSView

class Calculator(PloneKSSView):
    @kssaction
    def calculate(self, num):
        answer = str(2 * int(num))          # get answer
        core = self.getCommandSet('core')
        core.replaceInnerHTML('#result', answer)
```

- Start KSS & rendering done via decorator
- 

### 18.3.4 Alternate Script 2

- Add new class as view:

```
<browser:page
  for="*"
  name="kssCalc"
  class=".calculator.Calculator"
  attribute="calculate"
  permission="zope2.View"
/>
```

- This is now bound as “kssCalc”, and works as earlier

## 18.4 KSS Cheat Sheet

---

### 18.4.1 Actions: Changing HTML

- `replaceInnerHTML`: Replace all children of the given node with the given content.
    - `html`: the html to insert
  - `insertHTMLAfter`: Add HTML after given node.
    - `html`: the html to insert
  - `deleteNode`: Delete the node.
- 

### 18.4.2 Actions: Attributes

- `setAttribute`: Sets a given HTML attribute of the node.
  - `name`: the attribute name.
  - `value`: the attribute value to set
- `setStyle`: Sets a given style element on the node.
  - `name`: the name of the style element.
  - `value`: the style element value to set

### 18.4.3 Actions: CSS Classes

- `addClass`: Add a class to the classes of the node.
    - `value`: the name of the class
  - `removeClass`: Remove a class from the classes of the node.
    - `value`: the name of the class
  - `toggleClass`: Toggle class on node.
    - `value`: the name of the class
- 

### 18.4.4 Actions: Form Elements

- `focus`: Focus the given node that is a form input.
- 

### 18.4.5 Actions: Debugging

- `error`: Throws an exception, when executed.
- `log`: Logs an informational message.
  - `message`
- `alert`: Javascript alert box.
  - `message`

#### 18.4.6 Parameter Providers: Forms

- `formVar(formname, varname)`
    - Produces the value of a given variable within a given form.
  - `currentFormVar(varname)`
    - Produces the value of a given variable within the current form.
- 

#### 18.4.7 Parameter Providers: Content

- `nodeAttr(attrname)`
    - Produces the value of a given html attribute of the selected node.
  - `nodeContent()`
    - Produces the textual content of the node. Newlines are converted to spaces.
- 

#### 18.4.8 Command Sets: Core

- Commands are things used by scripts
- core command set are the same as the “actions”
  - `setAttribute, addClass, etc.`
  - except when called from code, first parameter is identifier
    - \* `core.setAttribute('#myitem', 'size', '30')`

### 18.4.9 Commands Sets: Zope + Plone

- Additional command sets for Zope and Plone specific stuff
    - Refreshing a viewlet
    - Refreshing a portlet
    - ... and more!
- 

### 18.4.10 Debugging KSS

- Use Firebug!
  - Turn on portal\_javascript debugging
  - Messages come out to Firebug console
- Try your scripts from URL directly

## 19 Pluggable Auth Service

## 19.1 Concepts

---

### 19.1.1 Plone PAS

- “Pluggable Authentication System”
    - Allows replacement of any part of security system
    - ie, can write new role-checking-system or user-storage
  - In Plone 2.5+
- 

### 19.1.2 PAS Concepts

- **Challenge:** “who are you?”
- **Credentials:** “I am Joel”
- **Authentication:** “Yes, you are”
- **Group assignment:** “You’re part of ...”
- **Role assignment:** “You act like...”
- **Enumeration:** listing of users
- **User adding/Editing of users**

### 19.1.3 PAS Plugins

- Examples
  - Users/groups/roles from SQL: SQLPASPlugin
  - LDAP: PAS LDAP Docs
  - Role for in-network people: AutoRole
  - Can't login twice at same time: NoDuplicateLogin
- Many others: Authentication Add-Ons

## 19.2 Only In-Network

---

### 19.2.1 Only In-Network Concept

- Only people from localhost can log in
    - Polite: hide “login\_form” link
    - Polite: don’t issue challenge
    - Real: refuse to authenticate
- 

### 19.2.2 In-Network Script

- zope-root/inNetwork.py:

```
request = context.REQUEST

return request['REMOTE_ADDR']== '127.0.0.1'
```
- 

### 19.2.3 Hide Login Form

- Change portal\_membership condition:

```
python:object.inNetwork() and member is None
```
- 

### 19.2.4 Script PAS Plugins

- Can write PAS plugins as PythonScripts
  - Add Scriptable Plugin called only\_inside
  - Name is unimportant

### 19.2.5 Learning the API

- Find interfaces in

Products/PluggableAuthService/interfaces

- Signature in `plugins.py`:

```
def challenge( request, response ):  
    """ ...  
    Returns True if it fired, False otherwise.
```

---

### 19.2.6 Refuse to Challenge

- `only_inside/challenge`:

```
##parameters=request,response  
if not context.inNetwork():  
    return True
```

---

### 19.2.7 Making it Work

- `only_inside` Interfaces tab
  - Add Challenge interface
- Activate
  - Turns on
  - Then move to first

### 19.2.8 Refusing to Authenticate

- `only_inside/authenticateCredentials:`

```
if not context.inNetwork():
    raise "OutsideOfNetwork", \
        "Out-of-network users cannot log in."
```

---

### 19.2.9 Making it Work

- `only_inside` Interfaces tab
    - Add Authenticate interface
  - Activate
    - Turns on
    - Then move to first
- 

### 19.2.10 Testing it Out

- No login button
- Won't challenge for needs-login
  - `http://site/change_ownership?userid=foo`
- Will refuse offered credentials
  - `http://site?__ac_name=admin&__ac_password=admin`

## 19.3 Free Role

---

### 19.3.1 Free Role Concept

- Give additional role to logged-in members
    - But only from localhost
- 

### 19.3.2 New Plugin

- Add Scriptable Plugin called `free_role`
- `free_role/getRolesForPrincipal`:

```
if context.inNetwork():  
    return ['Manager']
```
- Add interface for Roles and activate
  - Order of activation not important

## 19.4 On-Disk Plugins

---

### 19.4.1 On-Disk Plugins

- Can write plugins on disk as a product
    - Can also export ZMI scriptable plugins
- 

### 19.4.2 On-Disk Plugins Details

- Just wrapper for add-forms and class
  - Subclasses `BasePlugin`
  - Use `classImplements` to implement interface
  - Actual method is the same
- `NoGoChallenger` is nice and simple
- <http://svn.plone.org/svn/collective/PASPlugins/NoGoChallenger/trunk/>

## 19.5 Exercises

### 19.5.1 Exercises

1. Write a new plugin. If the user is coming from inside our network, this should ignore whatever username and password is provided for authentication, and log the user in as “admin”.
    - What part of the process is this?
    - How could we make this happen only if we can’t find the user?
- 

### 19.5.2 Exercise Answers

1. Plugin `simple_pass/authenticateCredentials`:

```
##parameters=credentials
if context.inNetwork():
    return ('admin', 'admin')
```

- Register and activate for `authenticate`

## **20 Plone 3 Fixes**

## 20.1 CM Fixes

---

### 20.1.1 Turn Off Inline Editing

- In `portal_kss`, turn off `at.kss`
    - This loses live validation on edit form, though
    - Refactoring may allow this to be easier in future
- 

### 20.1.2 Multipage Wizards

- AJAX schematas are pretty and fast
    - But don't reload fieldset during switches
    - One schemata can't rely on another
- 

### 20.1.3 Multipage Wizard Fix

- Marker interface `IMultiPageSchema` can be added to content class:

```
<five:implements
  class="Products.ATContentTypes.content.folder.ATFolder"
  interface=
  "Products.Archetypes.interfaces.IMultiPageSchema" />
```

## 20.2 Security Fixes

---

### 20.2.1 Adding Roles to Sharing

- Require on-disk Python class & ZCML registration
  - See examples in `plone/app/workflow/localroles.py` and `plone/app/workflow/configure.zcml`
    - Member, Manager, and Owner are just commented out
- 

### 20.2.2 Adding Roles to Sharing

- `localroles.py`:

```
from plone.app.workflow.interfaces import ISharingPageRole
from zope.interface import implements

class ModeratorRole(object):
    implements(ISharingPageRole)

    title = "Can moderate"
    required_permission = None
    # Or can name permission required to assign -- nice!
```

### 20.2.3 Adding Roles to Sharing II

- `configure.zcml`:

```
<utility
  name="Moderator"
  factory=".localroles.ModeratorRole"
/>
```

- These don't add role itself, just to sharing
    - Still should add in GS or via ZMI
- 

### 20.2.4 Old-Style Messages

```
request.redirect('/foo?portal_status_message=Hello')
```

- No way to have more than one status message
  - Ugly URLs
    - People may bookmark and be confused
  - Not everything can appear in URL
  - Still works, though!
- 

### 20.2.5 New-Style Messages

```
from Products.statusmessages.interfaces import IStatusMessage
IStatusMessage(request).addStatusMessage(message, type=type)
```

- Typical type info

## 21 Building Content Rules

## 21.1 Overview

### Overview

- Rules
- Actions

## 21.2 Building an Action

---

### 21.2.1 Action Overview

- Similar to building a new portlet
    - Interface for data model
    - Action for the ping logic
    - Executor for the action
      - \* Action and Executor could be same class
- 

### 21.2.2 Interface

- ping.py:

```
class IPingAction(Interface):
    """Definition of the config for ping"""

    url = schema.TextLine(title=u"URL",
                          description=u"URL to ping",
                          required=True)
```

### 21.2.3 Action

- ping.py (cont'd):

```
class PingAction(SimpleItem):
    """The implementation of the action defined before"""
    implements(IPingAction, IRuleElementData)

    url = ''

    element = 'plone.actions.Ping'

    @property
    def summary(self):
        return "Ping %s" % self.url
```

---

### 21.2.4 Executor

- ping.py (cont'd):

```
class PingActionExecutor(object):
    """The executor for this action."""
    implements(IExecutable)
    adapts(Interface, IPingAction, Interface)

    def __init__(self, context, element, event):
        self.context = context
        self.element = element
        self.event = event

    def __call__(self):
        url = self.element.url
        return True
```

### 21.2.5 Add Form

- ping.py (cont'd):

```
class PingAddForm(AddForm):
    """ An add form for the ping action """

    form_fields = form.FormFields(IPingAction)
    label = u"Add Ping Action"
    description = u"A ping action pings by getting URL."
    form_name = u"Configure element"

    def create(self, data):
        a = PingAction()
        form.applyChanges(a, self.form_fields, data)
        return a
```

---

### 21.2.6 Edit Form

- ping.py (cont'd):

```
class PingEditForm(EditForm):
    """ An edit form for the ping action """

    form_fields = form.FormFields(IPingAction)
    label = u"Edit Ping Action"
    description = u"A ping action pings by getting URL."
    form_name = u"Configure element"
```

---

### 21.2.7 ZCML Wiring

- First, register adapter in `configure.zcml`:

```
<adapter factory=".ping.PingActionExecutor" />
```

### 21.2.8 ZCML Wiring II

- Then, wire add/edit pages in `configure.zcml`:

```
<browser:page
  for="plone.app.contentrules.browser.interfaces. \
      IRuleActionAdding"
  name="plone.actions.Ping"
  class=".ping.PingAddForm"
  permission="cmf.ManagePortal" />

<browser:page
  for=".ping.IPingAction"
  name="edit"
  class=".ping.PingEditForm"
  permission="cmf.ManagePortal" />
```

---

### 21.2.9 ZCML Wiring III

- Last, register the action itself:

```
<plone:ruleAction
  name="plone.actions.Ping"
  title="Ping"
  description="Ping a particular server"
  for="*"
  event="*"
  addview="plone.actions.Ping"
  editview="edit"
/>
```

## **22 Internationalization**

## 22.1 Concepts

### Overview

- i18n of templates
  - What developers do
  - What translators do
- i18n of content

## 22.2 **Template i18n**

---

### 22.2.1 **Template i18n**

- Plone ships 50+ languages
    - Some perfect, some good start
    - Includes R-to-L support
- 

### 22.2.2 **Your Template**

```
<p>Welcome to Plone.</p>
```

```

```

```
<p>There have been over
  <span tal:content="context/download">
    100,000</span> downloads of Plone.
</p>
```

```
<p>Visit <a href="about">About Plone</a>
  for more info.
</p>
```

### 22.2.3 Your Template: Need to Do

```
<p>Welcome to Plone.</p> translate

 translate alt text

<p>There have been over
  <span tal:content="context/download">
    100,000</span> downloads of Plone.
</p> translate, move # as needed

<p>Visit <a href="about">About Plone</a>
  for more info. translate, keep link, move
</p>
```

### 22.2.4 Case 1: “Welcome to Plone”

- ```
<p>Welcome to Plone.</p> translate
```
- i18n:translate marks as needing translation
 

```
<p i18n:translate="">Welcome to Plone.</p>
```

    - This uses “Welcome to Plone” as string-to-translate

### 22.2.5 Case 1: “Welcome...” (2)

- Some words have multiple meanings, eg “set”
  - A collection of things
  - To establish a choice

```
<p i18n:translate="">Set</p>
```
- Unclear which we mean

### 22.2.6 Case 1: "Welcome..." (3)

- Can pick a "message id" (msgid)

```
<p i18n:translate="set-a-setting">Set</p>
```

- Translators now know what meaning to use
- 

### 22.2.7 Case 2: Alt Text of Image

```
 translate
```

- i18n:attributes:

```

```

- Marks "alt" as needing translation
- 

### 22.2.8 Case 2: Alt Text (2)

- Example with two attributes:

```

```

- Mark both as needing translation:

```

```

### 22.2.9 Case 2: Alt Text (2)

- Can use msgids:

```

```

---

### 22.2.10 Case 3: Dynamic Content

```
<p>There have been over
  <span tal:content="context/download">
    100,000</span> downloads of Plone.
</p> translate, move # as needed
```

- Need to translate sentence
    - # itself isn't translated
- 

### 22.2.11 Case 3: Dynamic Content (2)

- But needs to go in the right place
  - Won't be the same in every language
    - \* "There have been over 10,000 downloads"
    - \* "There have over 10,000 downloads been"
    - \* "Over 10,000 have been downloads"

### 22.2.12 Case 3: Dynamic Content (3)

- Want translators to see:

There have been `${count}` downloads of Plone

- `i18n:name` to name the special part:

```
<p i18n:translate="">There have been over  
  <span tal:content="here/download_count"  
    i18n:name="count">100,000</span>  
  downloads of Plone.  
</p>
```

- Whole thing is marked with `i18n:translate`
- 

### 22.2.13 Case 3: Dynamic Content (4)

- `i18n:name` isn't related to how things work:

```
<span tal:content="here/download_count"  
  i18n:name="count">100,000</span>
```

- `download_count` is real method
- `count` is name for translators
- Must be unique only in `i18n:translate` tag
  - Other things can be called `count`

**22.2.14 Case 4: Combining Ideas**

```
<p>Visit <a href="about">About Plone</a>
  for more info. translate, keep link, move
</p>
```

- About Plone must be movable and translated:

```
<p i18n:translate="">Please visit
  <span i18n:name="about-plone">
    <a href="about" i18n:translate="">
      About Plone</a>
    </span> for more info.</p>
```

**22.2.15 i18n Domain**

- `i18n:domain` defines “domain” for translations
  - Different apps might translate differently
  - Use your app or company name

**22.2.16 Complete Template**

```
<html i18n:domain="myprod"><body>
<p i18n:translate="">Welcome to Plone.</p>

<p i18n:translate="">There have been over
  <span tal:content="here/download_count"
    i18n:name="count">100,000</span>
  downloads of Plone.</p>
<p i18n:translate="">Please visit
  <span i18n:name="about-plone">
    <a href="about" i18n:translate="">About Plone</a>
  </span> for more info.</p>
</body></html>
```

## 22.3 Translating

---

### 22.3.1 Creating Template

```
i18ndude rebuild-pot
  --pot myprod.pot  output file location
  --create myprod   # your domain
  *.pt             path to templates
```

---

### 22.3.2 POT File

- myprod.pot

```
"Language-Code: en"
"Language-Name: English"
"Domain: myprod"

#. Default: "About Plone"
#: myzpt.pt
msgid "About Plone"
msgstr ""

#. Default: "Please visit ${about-plone} for more info."
#: myzpt.pt
msgid "Please visit ${about-plone} for more info."
msgstr ""

#. Default: "Plone Icon"
#: myzpt.pt
msgid "Plone Icon"
msgstr ""
```

### 22.3.3 POTs and POs

- POT is the **template**
  - PO is copied from that
    - And changed for a language
    - One PO per language you want
- 

### 22.3.4 PO File Translate

- myprod-pl.po

```
"Language-Code: pl"  
"Language-Name: Pig Latin"  
"Domain: myprod"
```

```
#. Default: "About Plone"  
#: myzpt.pt  
msgid "About Plone"  
msgstr "Aboutay Lonipay"
```

```
#. Default: "Please visit ${about-plone} for more info."  
#: myzpt.pt  
msgid "Please visit ${about-plone} for more info."  
msgstr "Leasipay isitvay ${about-plone} orfay ormay infoay."
```

```
#. Default: "Plone Icon"  
#: myzpt.pt  
msgid "Plone Icon"  
msgstr "Lonipay Iconay"
```

### 22.3.5 Help for Translators

- We're using standard machinery
  - So use existing editors, checkers, etc
- i18ndude has features to find
  - Missing `i18n: translates`
  - Text that isn't translated

## 22.4 Translating Scripts

### Translating Scripts

- In your product, create a “MessageFactory”:

```
from zope.i18nmessageid import MessageFactory
MyProdMessageFactory = MessageFactory('MyProduct')
```

- In scripts, import (by convention, as \_):

```
from MyProduct import MyProdMessageFactory as _
print _(u"This can be translated.")
```

## 22.5 Translating Content

---

### 22.5.1 Translating Content

- We don't do automatically
    - Machine generated translation can be poor
  - We give you tools to help humans
    - LinguaPlone is the most popular
- 

### 22.5.2 LinguaPlone

- Choose languages your site will use
    - Site Setup -> Language Tool
  - For a piece of content, can manage translations
  - Can translate to each language
- 

### 22.5.3 LinguaPlone Options

- How does it figure out "your" language
  - Cookies, forms, browsers, etc
- Does it use flags? Or names?
  - Flags are pretty but tricky
    - \* What to show for English?
      - US?
      - Britain?
      - India?

#### 22.5.4 **Language-Independent**

- Some fields never will be translated
  - People's names
  - Birth dates
  - Phone numbers
- Tagged value: `languageIndependent`

## 23 Unit Tests

## 23.1 Overview

### Overview

- Running unit tests
- Writing unit tests
- Test frameworks

## 23.2 Running Unit Tests

### Running

```
zopectl test -m Products.ProductName
```

- -D option for postmortem

## 23.3 Writing Unit Tests

---

### 23.3.1 Setup

- Must have PloneTestCase
    - Included with Plone 2.5+
  - Create tests package
    - (Don't forget `__init__.py`!)
- 

### 23.3.2 Base Class

- `tests/base.py`

```
from Testing import ZopeTestCase
from Products.PloneTestCase.PloneTestCase import \
    PloneTestCase, setupPloneSite

ZopeTestCase.installProduct('MyProduct')

setupPloneSite(products=('MyProduct',))

class MyProductTestCase(PloneTestCase):
    "Base class for your product's tests"
```

### 23.3.3 Sample Test

- tests/test\_setup.py:

```
from base import MyProductTestCase

class TestProductInstall(MyProductTestCase):

    def testTypesInstalled(self):
        for t in ['Foo']:
            self.failUnless(t in
                self.portal.portal_types.objectIds(),
                '%s content type not installed' % t)
```

---

### 23.3.4 Test Suite

- tests/test\_setup.py:

```
def test_suite():
    from unittest import TestSuite, makeSuite
    suite = TestSuite()
    suite.addTest(makeSuite(TestProductInstall))
    return suite
```

---

### 23.3.5 Scaffolding

- ZODB is aborted after each test
  - So cleanup is done for us
- Need to be done before tests?
  - afterSetUp
- Extra cleanup needed?
  - beforeTearDown

### 23.3.6 Test Functions

- `failUnless(expr[, msg])`
    - Ensure `expr` is true
  - `assertEqual(expr1, expr2[, msg])`
    - Ensure `expr1` is equal to `expr2`
  - `fail([msg])`
    - Fail
    - Useful in condition
- 

### 23.3.7 Helper Attributes

- `self.portal`
    - Your portal
  - `self.folder`
    - Member folder for “you”
- 

### 23.3.8 Helper Methods

- `self.logout()`
- `self.login([username])`
- `self.setRoles(roles)`
  - Pass in list of roles for you
- `self.setPermissions(perms)`
  - Pass in list of perms for you

## 23.4 DocTests

---

### 23.4.1 About DocTests

- Tests embedded into documentation files
    - Natural way of explaining + testing
  - Tests in top-level as “foo.txt”
- 

### 23.4.2 Sample DocTest

- test.txt:

Let's test some basic math:

```
>>> 1 + 1
2
```

This won't work:

```
>>> 1 + 1
3
```

### 23.4.3 DocTest

- tests/test\_doctests.py:

```
import os, sys
import doctest
import unittest
from base import MyProductTestCase
from Testing.ZopeTestCase import FunctionalDocFileSuite

OPTIONFLAGS = ( doctest.REPORT_ONLY_FIRST_FAILURE |
                 doctest.ELLIPSIS |
                 doctest.NORMALIZE_WHITESPACE)
```

---

### 23.4.4 DocTest (2)

- tests/test\_doctests.py:

```
def test_suite():
    filenames = ['test.txt']
    return unittest.TestSuite(
        [ FunctionalDocFileSuite(
            os.path.basename(filename),
            optionflags=OPTIONFLAGS,
            package='Products.Tester',
            test_class=MyProductTestCase)
          for filename in filenames ]
    )
```

---

### 23.4.5 DocTest Tips

- Still have
  - afterSetUp and beforeTearDown
  - access to self and helper methods

## 23.5 Road Ahead

### Road Ahead

- In-code doctests
- `zope.testbrowser` and `zope.testrecorder`