

---

# Setuptools Documentation

*Release 46.1.3*

**The fellowship of the packaging**

**Apr 06, 2020**



<b>1</b>	<b>Building and Distributing Packages with Setuptools</b>	<b>3</b>
1.1	Developer's Guide . . . . .	5
1.2	Command Reference . . . . .	25
1.3	Configuring setup() using setup.cfg files . . . . .	32
1.4	Extending and Reusing Setuptools . . . . .	37
<b>2</b>	<b>Package Discovery and Resource Access using pkg_resources</b>	<b>41</b>
2.1	Overview . . . . .	42
2.2	API Reference . . . . .	43
<b>3</b>	<b>Supporting both Python 2 and Python 3 with Setuptools</b>	<b>67</b>
3.1	Using 2to3 . . . . .	67
3.2	Distributing Python 3 modules . . . . .	68
3.3	Advanced features . . . . .	68
<b>4</b>	<b>Development on Setuptools</b>	<b>69</b>
4.1	Developer's Guide for Setuptools . . . . .	69
4.2	The Internal Structure of Python Eggs . . . . .	72
4.3	Release Process . . . . .	81
<b>5</b>	<b>Roadmap</b>	<b>83</b>
<b>6</b>	<b>Easy Install</b>	<b>85</b>
6.1	Using "Easy Install" . . . . .	86
6.2	Reference Manual . . . . .	95
<b>7</b>	<b>History</b>	<b>101</b>
7.1	v46.1.3 . . . . .	101
7.2	v46.1.2 . . . . .	101
7.3	v46.1.1 . . . . .	101
7.4	v46.1.0 . . . . .	101
7.5	v44.1.0 . . . . .	102
7.6	v46.0.0 . . . . .	102
7.7	v45.3.0 . . . . .	102
7.8	v45.2.0 . . . . .	102
7.9	v45.1.0 . . . . .	103
7.10	v45.0.0 . . . . .	103

7.11	v44.0.0	103
7.12	v43.0.0	103
7.13	v42.0.2	103
7.14	v42.0.1	104
7.15	v42.0.0	104
7.16	v41.6.0	104
7.17	v41.5.1	104
7.18	v41.5.0	105
7.19	v41.4.0	105
7.20	v41.3.0	105
7.21	v41.2.0	105
7.22	v41.1.0	105
7.23	v41.0.1	106
7.24	v41.0.0	106
7.25	v40.9.0	106
7.26	v40.8.0	107
7.27	v40.7.3	107
7.28	v40.7.2	107
7.29	v40.7.1	107
7.30	v40.7.0	107
7.31	v40.6.3	108
7.32	v40.6.2	108
7.33	v40.6.1	108
7.34	v40.6.0	108
7.35	v40.5.0	109
7.36	v40.4.3	109
7.37	v40.4.2	109
7.38	v40.4.1	109
7.39	v40.4.0	109
7.40	v40.3.0	109
7.41	v40.2.0	110
7.42	v40.1.1	110
7.43	v40.1.0	110
7.44	v40.0.0	110
7.45	v39.2.0	111
7.46	v39.1.0	111
7.47	v39.0.1	112
7.48	v39.0.0	112
7.49	v38.7.0	112
7.50	v38.6.1	112
7.51	v38.6.0	112
7.52	v38.5.2	112
7.53	v38.5.1	113
7.54	v38.5.0	113
7.55	v38.4.1	113
7.56	v38.4.0	113
7.57	v38.3.0	113
7.58	v38.2.5	113
7.59	v38.2.4	113
7.60	v38.2.3	114
7.61	v38.2.2	114
7.62	v38.2.1	114
7.63	v38.2.0	114
7.64	v38.1.0	114

7.65	v38.0.0	114
7.66	v37.0.0	114
7.67	v36.8.0	115
7.68	v36.7.3	115
7.69	v36.7.2	115
7.70	v36.7.1	115
7.71	v36.7.0	115
7.72	v36.6.1	115
7.73	v36.6.0	115
7.74	v36.5.0	116
7.75	v36.4.0	116
7.76	v36.3.0	116
7.77	v36.2.7	116
7.78	v36.2.6	116
7.79	v36.2.5	117
7.80	v36.2.4	117
7.81	v36.2.3	117
7.82	v36.2.2	117
7.83	v36.2.1	117
7.84	v36.2.0	117
7.85	v36.1.1	118
7.86	v36.1.0	118
7.87	v36.0.1	118
7.88	v36.0.0	118
7.89	v35.0.2	118
7.90	v35.0.1	118
7.91	v35.0.0	119
7.92	v34.4.1	119
7.93	v34.4.0	119
7.94	v34.3.3	119
7.95	v34.3.2	119
7.96	v34.3.1	119
7.97	v34.3.0	120
7.98	v34.2.0	120
7.99	v34.1.1	120
7.100	v34.1.0	120
7.101	v34.0.3	120
7.102	v34.0.2	120
7.103	v34.0.1	121
7.104	v34.0.0	121
7.105	v33.1.1	121
7.106	v33.1.0	121
7.107	v33.0.0	121
7.108	v32.3.1	122
7.109	v32.3.0	122
7.110	v32.2.0	122
7.111	v32.1.3	122
7.112	v32.1.2	122
7.113	v32.1.1	122
7.114	v32.1.0	122
7.115	v32.0.0	123
7.116	v31.0.1	123
7.117	v31.0.0	123
7.118	v30.4.0	123

7.119 v30.3.0	123
7.120 v30.2.1	123
7.121 v30.2.0	124
7.122 v30.1.0	124
7.123 v30.0.0	124
7.124 v29.0.1	124
7.125 v29.0.0	124
7.126 v28.8.0	124
7.127 v28.7.1	125
7.128 v28.7.0	125
7.129 v28.6.1	125
7.130 v28.6.0	125
7.131 v28.5.0	125
7.132 v28.4.0	125
7.133 v28.3.0	126
7.134 v28.1.0	126
7.135 v28.0.0	126
7.136 v27.3.1	126
7.137 v27.3.0	126
7.138 v27.2.0	127
7.139 v27.1.2	127
7.140 v27.1.1	127
7.141 v27.1.0	127
7.142 v27.0.0	127
7.143 v26.1.1	128
7.144 v26.1.0	128
7.145 v26.0.0	128
7.146 v25.4.0	128
7.147 v25.3.0	128
7.148 v25.2.0	129
7.149 v25.1.6	129
7.150 v25.1.5	129
7.151 v25.1.4	129
7.152 v25.1.3	129
7.153 v25.1.2	129
7.154 v25.1.1	130
7.155 v25.1.0	130
7.156 v25.0.2	130
7.157 v25.0.1	130
7.158 v25.0.0	130
7.159 v24.3.1	131
7.160 v24.3.0	131
7.161 v24.2.1	131
7.162 v24.2.0	131
7.163 v24.1.1	131
7.164 v24.1.0	131
7.165 v24.0.3	131
7.166 v24.0.2	132
7.167 v24.0.1	132
7.168 v24.0.0	132
7.169 v23.2.1	132
7.170 v23.1.0	132
7.171 v23.0.0	132
7.172 v22.0.5	133

7.173 v22.0.4 . . . . .	133
7.174 v22.0.3 . . . . .	133
7.175 v22.0.2 . . . . .	133
7.176 v22.0.1 . . . . .	133
7.177 v22.0.0 . . . . .	133
7.178 v21.2.2 . . . . .	133
7.179 v21.2.1 . . . . .	134
7.180 v21.2.0 . . . . .	134
7.181 v21.1.0 . . . . .	134
7.182 v21.0.0 . . . . .	134
7.183 v20.10.0 . . . . .	134
7.184 v20.9.0 . . . . .	134
7.185 v20.8.1 . . . . .	135
7.186 v20.8.0 . . . . .	135
7.187 v20.7.0 . . . . .	135
7.188 v20.6.8 . . . . .	135
7.189 v20.6.7 . . . . .	135
7.190 v20.6.6 . . . . .	135
7.191 v20.6.0 . . . . .	136
7.192 20.5 . . . . .	136
7.193 20.4 . . . . .	136
7.194 20.3.1 . . . . .	136
7.195 20.3 . . . . .	136
7.196 20.2.2 . . . . .	136
7.197 20.2.1 . . . . .	137
7.198 20.2 . . . . .	137
7.199 20.1.1 . . . . .	137
7.200 20.1 . . . . .	137
7.201 20.0 . . . . .	137
7.202 19.7 . . . . .	137
7.203 19.6.2 . . . . .	138
7.204 19.6.1 . . . . .	138
7.205 19.6 . . . . .	138
7.206 19.5 . . . . .	138
7.207 19.4.1 . . . . .	138
7.208 19.4 . . . . .	139
7.209 19.3 . . . . .	139
7.210 19.2 . . . . .	139
7.211 19.1.1 . . . . .	139
7.212 19.1 . . . . .	139
7.213 19.0 . . . . .	139
7.214 18.8.1 . . . . .	140
7.215 18.8 . . . . .	140
7.216 18.7.1 . . . . .	140
7.217 18.7 . . . . .	140
7.218 18.6.1 . . . . .	140
7.219 18.6 . . . . .	141
7.220 18.5 . . . . .	141
7.221 18.4 . . . . .	141
7.222 18.3.2 . . . . .	141
7.223 18.3.1 . . . . .	141
7.224 18.3 . . . . .	141
7.225 18.2 . . . . .	142
7.226 18.1 . . . . .	142

7.227 18.0.1 . . . . .	142
7.228 18.0 . . . . .	142
7.229 17.1.1 . . . . .	142
7.230 17.1 . . . . .	143
7.231 17.0 . . . . .	143
7.232 16.0 . . . . .	143
7.233 15.2 . . . . .	143
7.234 15.1 . . . . .	143
7.235 15.0 . . . . .	143
7.236 14.3.1 . . . . .	144
7.237 14.3 . . . . .	144
7.238 14.2 . . . . .	144
7.239 14.1.1 . . . . .	144
7.240 14.1 . . . . .	144
7.241 14.0 . . . . .	144
7.242 13.0.2 . . . . .	145
7.243 13.0.1 . . . . .	145
7.244 13.0 . . . . .	145
7.245 12.4 . . . . .	145
7.246 12.3 . . . . .	145
7.247 12.2 . . . . .	146
7.248 12.1 . . . . .	146
7.249 12.0.5 . . . . .	146
7.250 12.0.4 . . . . .	146
7.251 12.0.3 . . . . .	146
7.252 12.0.2 . . . . .	146
7.253 12.0.1 . . . . .	146
7.254 12.0 . . . . .	147
7.255 11.3.1 . . . . .	147
7.256 11.3 . . . . .	147
7.257 11.2 . . . . .	147
7.258 11.1 . . . . .	147
7.259 11.0 . . . . .	148
7.260 10.2.1 . . . . .	148
7.261 10.2 . . . . .	148
7.262 10.1 . . . . .	148
7.263 10.0.1 . . . . .	148
7.264 10.0 . . . . .	148
7.265 9.1 . . . . .	149
7.266 9.0.1 . . . . .	149
7.267 9.0 . . . . .	149
7.268 8.4 . . . . .	149
7.269 8.3 . . . . .	149
7.270 8.2.1 . . . . .	149
7.271 8.2 . . . . .	149
7.272 8.1 . . . . .	150
7.273 8.0.4 . . . . .	150
7.274 8.0.3 . . . . .	150
7.275 8.0.2 . . . . .	150
7.276 8.0.1 . . . . .	150
7.277 8.0 . . . . .	150
7.278 7.0 . . . . .	151
7.279 6.1 . . . . .	151
7.280 6.0.2 . . . . .	151



7.281 6.0.1 . . . . .	151
7.282 6.0 . . . . .	151
7.283 5.8 . . . . .	152
7.284 5.7 . . . . .	152
7.285 5.6 . . . . .	152
7.286 5.5.1 . . . . .	152
7.287 5.5 . . . . .	152
7.288 5.4.2 . . . . .	153
7.289 5.4.1 . . . . .	153
7.290 5.4 . . . . .	153
7.291 5.3 . . . . .	153
7.292 5.2 . . . . .	153
7.293 5.1 . . . . .	153
7.294 5.0.2 . . . . .	154
7.295 5.0.1 . . . . .	154
7.296 5.0 . . . . .	154
7.297 3.7.1 and 3.8.1 and 4.0.1 . . . . .	154
7.298 4.0 . . . . .	154
7.299 3.8 . . . . .	154
7.300 3.7 . . . . .	154
7.301 3.6 . . . . .	155
7.302 3.5.2 . . . . .	155
7.303 3.5.1 . . . . .	155
7.304 3.5 . . . . .	155
7.305 3.4.4 . . . . .	155
7.306 3.4.3 . . . . .	155
7.307 3.4.2 . . . . .	156
7.308 3.4.1 . . . . .	156
7.309 3.4 . . . . .	156
7.310 3.3 . . . . .	156
7.311 3.2 . . . . .	156
7.312 3.1 . . . . .	156
7.313 3.0.2 . . . . .	157
7.314 3.0.1 . . . . .	157
7.315 3.0 . . . . .	157
7.316 2.2 . . . . .	157
7.317 2.1.2 . . . . .	158
7.318 2.1.1 . . . . .	158
7.319 2.1 . . . . .	158
7.320 2.0.2 . . . . .	158
7.321 2.0.1 . . . . .	158
7.322 2.0 . . . . .	158
7.323 1.4.2 . . . . .	159
7.324 1.4.1 . . . . .	159
7.325 1.4 . . . . .	159
7.326 1.3.2 . . . . .	159
7.327 1.3.1 . . . . .	160
7.328 1.3 . . . . .	160
7.329 1.2 . . . . .	160
7.330 1.1.7 . . . . .	160
7.331 1.1.6 . . . . .	160
7.332 1.1.5 . . . . .	160
7.333 1.1.4 . . . . .	161
7.334 1.1.3 . . . . .	161

7.335 1.1.2 . . . . .	161
7.336 1.1.1 . . . . .	161
7.337 1.1 . . . . .	161
7.338 1.0 . . . . .	161
7.339 0.9.8 . . . . .	162
7.340 0.9.7 . . . . .	162
7.341 0.9.6 . . . . .	162
7.342 0.9.5 . . . . .	162
7.343 0.9.4 . . . . .	163
7.344 0.9.3 . . . . .	163
7.345 0.9.2 . . . . .	163
7.346 0.9.1 . . . . .	163
7.347 0.9 . . . . .	163
7.348 0.8 . . . . .	163
7.349 0.7.8 . . . . .	163
7.350 0.7.7 . . . . .	164
7.351 0.7.6 . . . . .	164
7.352 0.7.5 . . . . .	164
7.353 0.7.4 . . . . .	164
7.354 0.7.3 . . . . .	164
7.355 0.7.2 . . . . .	164
7.356 0.7.1 . . . . .	165
7.357 0.7 . . . . .	165
7.358 0.7b4 . . . . .	165
7.359 0.6.49 . . . . .	165
7.360 0.6.48 . . . . .	165
7.361 0.6.47 . . . . .	165
7.362 0.6.46 . . . . .	166
7.363 0.6.45 . . . . .	166
7.364 0.6.44 . . . . .	166
7.365 0.6.43 . . . . .	166
7.366 0.6.42 . . . . .	166
7.367 0.6.41 . . . . .	166
7.368 0.6.40 . . . . .	167
7.369 0.6.39 . . . . .	167
7.370 0.6.38 . . . . .	167
7.371 0.6.37 . . . . .	167
7.372 0.6.36 . . . . .	167
7.373 0.6.35 . . . . .	167
7.374 0.6.34 . . . . .	168
7.375 0.6.33 . . . . .	168
7.376 0.6.32 . . . . .	168
7.377 0.6.31 . . . . .	168
7.378 0.6.30 . . . . .	169
7.379 0.6.29 . . . . .	169
7.380 0.6.28 . . . . .	170
7.381 0.6.27 . . . . .	170
7.382 0.6.26 . . . . .	170
7.383 0.6.25 . . . . .	170
7.384 0.6.24 . . . . .	170
7.385 0.6.23 . . . . .	171
7.386 0.6.21 . . . . .	171
7.387 0.6.20 . . . . .	171
7.388 0.6.19 . . . . .	171

7.389 0.6.18 . . . . .	171
7.390 0.6.17 . . . . .	172
7.391 0.6.16 . . . . .	172
7.392 0.6.15 . . . . .	172
7.393 0.6.14 . . . . .	172
7.394 0.6.13 . . . . .	172
7.395 0.6.12 . . . . .	173
7.396 0.6.11 . . . . .	173
7.397 0.6.10 . . . . .	173
7.398 0.6.9 . . . . .	173
7.399 0.6.8 . . . . .	174
7.400 0.6.7 . . . . .	174
7.401 0.6.6 . . . . .	175
7.402 0.6.5 . . . . .	175
7.403 0.6.4 . . . . .	175
7.404 0.6.3 . . . . .	175
7.405 0.6.2 . . . . .	175
7.406 0.6.1 . . . . .	176
7.407 0.6 . . . . .	176
7.408 0.6c9 . . . . .	177
7.409 0.6c7 . . . . .	178
7.410 0.6c6 . . . . .	178
7.411 0.6c5 . . . . .	179
7.412 0.6c4 . . . . .	179
7.413 0.6c3 . . . . .	179
7.414 0.6c2 . . . . .	180
7.415 0.6c1 . . . . .	180
7.416 0.6b4 . . . . .	180
7.417 0.6b3 . . . . .	181
7.418 0.6b2 . . . . .	181
7.419 0.6b1 . . . . .	181
7.420 0.6a11 . . . . .	182
7.421 0.6a10 . . . . .	182
7.422 0.6a9 . . . . .	183
7.423 0.6a8 . . . . .	184
7.424 0.6a7 . . . . .	184
7.425 0.6a6 . . . . .	185
7.426 0.6a5 . . . . .	185
7.427 0.6a3 . . . . .	185
7.428 0.6a2 . . . . .	185
7.429 0.6a1 . . . . .	186
7.430 0.5a12 . . . . .	187
7.431 0.5a11 . . . . .	187
7.432 0.5a10 . . . . .	187
7.433 0.5a9 . . . . .	187
7.434 0.5a8 . . . . .	188
7.435 0.5a7 . . . . .	188
7.436 0.5a6 . . . . .	189
7.437 0.5a5 . . . . .	189
7.438 0.5a4 . . . . .	189
7.439 0.5a3 . . . . .	190
7.440 0.5a2 . . . . .	190
7.441 0.5a1 . . . . .	190
7.442 0.4a4 . . . . .	190

7.443 0.4a3 . . . . .	190
7.444 0.4a2 . . . . .	191
7.445 0.4a1 . . . . .	191
7.446 0.3a4 . . . . .	191
7.447 0.3a3 . . . . .	191
7.448 0.3a2 . . . . .	192
7.449 0.3a1 . . . . .	192
<b>8 Credits</b>	<b>193</b>
<b>Index</b>	<b>195</b>

Setuptools is a fully-featured, actively-maintained, and stable library designed to facilitate packaging Python projects, where packaging includes:

- Python package and module definitions
- Distribution package metadata
- Test hooks
- Project installation
- Platform-specific details
- Python 3 support

Documentation content:



---

## Building and Distributing Packages with Setuptools

---

Setuptools is a collection of enhancements to the Python `distutils` that allow developers to more easily build and distribute Python packages, especially ones that have dependencies on other packages.

Packages built and distributed using `setuptools` look to the user like ordinary Python packages based on the `distutils`.

Feature Highlights:

- Create [Python Eggs](#) - a single-file importable distribution format
- Enhanced support for accessing data files hosted in zipped packages.
- Automatically include all packages in your source tree, without listing them individually in `setup.py`
- Automatically include all relevant files in your source distributions, without needing to create a `MANIFEST.in` file, and without having to force regeneration of the `MANIFEST` file when your source tree changes.
- Automatically generate wrapper scripts or Windows (console and GUI) `.exe` files for any number of “main” functions in your project. (Note: this is not a `py2exe` replacement; the `.exe` files rely on the local Python installation.)
- Transparent Cython support, so that your `setup.py` can list `.pyx` files and still work even when the end-user doesn’t have Cython installed (as long as you include the Cython-generated C in your source distribution)
- Command aliases - create project-specific, per-user, or site-wide shortcut names for commonly used commands and options
- Deploy your project in “development mode”, such that it’s available on `sys.path`, yet can still be edited directly from its source checkout.
- Easily extend the `distutils` with new commands or `setup()` arguments, and distribute/reuse your extensions for multiple projects, without copying code.
- Create extensible applications and frameworks that automatically discover extensions, using simple “entry points” declared in a project’s setup script.
- Full support for PEP 420 via `find_namespace_packages()`, which is also backwards compatible to the existing `find_packages()` for Python `>= 3.3`.

## Table of Contents

- *Building and Distributing Packages with Setuptools*
  - *Developer’s Guide*
    - \* *Installing setuptools*
    - \* *Basic Use*
      - *Specifying Your Project’s Version*
    - \* *New and Changed setup() Keywords*
      - *Using find\_packages()*
      - *find\_namespace\_packages()*
    - \* *Automatic Script Creation*
      - *“Eggsecutable” Scripts*
    - \* *Declaring Dependencies*
      - *Dependencies that aren’t in PyPI*
      - *Declaring “Extras” (optional features with their own dependencies)*
      - *Declaring platform specific dependencies*
    - \* *Including Data Files*
      - *Accessing Data Files at Runtime*
      - *Non-Package Data Files*
      - *Automatic Resource Extraction*
    - \* *Extensible Applications and Frameworks*
      - *Dynamic Discovery of Services and Plugins*
      - *Defining Additional Metadata*
    - \* *“Development Mode”*
    - \* *Distributing a setuptools-based project*
      - *Setting the zip\_safe flag*
      - *Namespace Packages*
      - *TRANSITIONAL NOTE*
      - *Tagging and “Daily Build” or “Snapshot” Releases*
      - *Generating Source Distributions*
      - *Making “Official” (Non-Snapshot) Releases*
      - *Distributing Extensions compiled with Cython*
  - *Command Reference*
    - \* *alias - Define shortcuts for commonly used commands*
    - \* *bdist\_egg - Create a Python Egg for the project*
    - \* *develop - Deploy the project source in “Development Mode”*



- \* *egg\_info* - Create egg metadata and set build tags
  - *Release Tagging Options*
  - *Other egg\_info Options*
  - *egg\_info Examples*
- \* *rotate* - Delete outdated distribution files
- \* *saveopts* - Save used options to a configuration file
  - *Configuration File Options*
- \* *setopt* - Set a distutils or setuptools option in a config file
- \* *test* - Build package and run a unittest suite
- \* *upload* - Upload source and/or egg distributions to PyPI
- *Configuring setup() using setup.cfg files*
  - \* *Using a src/ layout*
  - \* *Specifying values*
    - *Metadata*
    - *Options*
  - \* *Configuration API*
- *Extending and Reusing Setuptools*
  - \* *Creating distutils Extensions*
    - *Adding Commands*
    - *Adding setup() Arguments*
    - *Customizing Distribution Options*
    - *Adding new EGG-INFO Files*
    - *Adding Support for Revision Control Systems*
  - \* *Mailing List and Bug Tracker*

## 1.1 Developer's Guide

### 1.1.1 Installing setuptools

To install the latest version of setuptools, use:

```
pip install --upgrade setuptools
```

Refer to [Installing Packages](#) guide for more information.

### 1.1.2 Basic Use

For basic use of setuptools, just import things from setuptools instead of the distutils. Here's a minimal setup script using setuptools:

```
from setuptools import setup, find_packages
setup(
    name="HelloWorld",
    version="0.1",
    packages=find_packages(),
)
```

As you can see, it doesn't take much to use setuptools in a project. Run that script in your project folder, alongside the Python packages you have developed.

Invoke that script to produce distributions and automatically include all packages in the directory where the setup.py lives. See the *Command Reference* section below to see what commands you can give to this setup script. For example, to produce a source distribution, simply invoke:

```
setup.py sdist
```

Of course, before you release your project to PyPI, you'll want to add a bit more information to your setup script to help people find or learn about your project. And maybe your project will have grown by then to include a few dependencies, and perhaps some data files and scripts:

```
from setuptools import setup, find_packages
setup(
    name="HelloWorld",
    version="0.1",
    packages=find_packages(),
    scripts=["say_hello.py"],

    # Project uses reStructuredText, so ensure that the docutils get
    # installed or upgraded on the target machine
    install_requires=["docutils>=0.3"],

    package_data={
        # If any package contains *.txt or *.rst files, include them:
        "": ["*.txt", "*.rst"],
        # And include any *.msg files found in the "hello" package, too:
        "hello": ["*.msg"],
    },

    # metadata to display on PyPI
    author="Me",
    author_email="me@example.com",
    description="This is an Example Package",
    keywords="hello world example examples",
    url="http://example.com/HelloWorld/", # project home page, if any
    project_urls={
        "Bug Tracker": "https://bugs.example.com/HelloWorld/",
        "Documentation": "https://docs.example.com/HelloWorld/",
        "Source Code": "https://code.example.com/HelloWorld/",
    },
    classifiers=[
        "License :: OSI Approved :: Python Software Foundation License"
    ]

    # could also include long_description, download_url, etc.
)
```

In the sections that follow, we'll explain what most of these setup() arguments do (except for the metadata ones), and the various ways you might use them in your own project(s).

## Specifying Your Project's Version

Setuptools can work well with most versioning schemes; there are, however, a few special things to watch out for, in order to ensure that setuptools and other tools can always tell what version of your package is newer than another version. Knowing these things will also help you correctly specify what versions of other projects your project depends on.

A version consists of an alternating series of release numbers and pre-release or post-release tags. A release number is a series of digits punctuated by dots, such as 2.4 or 0.5. Each series of digits is treated numerically, so releases 2.1 and 2.1.0 are different ways to spell the same release number, denoting the first subrelease of release 2. But 2.10 is the *tenth* subrelease of release 2, and so is a different and newer release from 2.1 or 2.1.0. Leading zeros within a series of digits are also ignored, so 2.01 is the same as 2.1, and different from 2.0.1.

Following a release number, you can have either a pre-release or post-release tag. Pre-release tags make a version be considered *older* than the version they are appended to. So, revision 2.4 is *newer* than revision 2.4c1, which in turn is newer than 2.4b1 or 2.4a1. Postrelease tags make a version be considered *newer* than the version they are appended to. So, revisions like 2.4-1 and 2.4p13 are newer than 2.4, but are *older* than 2.4.1 (which has a higher release number).

A pre-release tag is a series of letters that are alphabetically before “final”. Some examples of prerelease tags would include alpha, beta, a, c, dev, and so on. You do not have to place a dot or dash before the prerelease tag if it’s immediately after a number, but it’s okay to do so if you prefer. Thus, 2.4c1 and 2.4.c1 and 2.4-c1 all represent release candidate 1 of version 2.4, and are treated as identical by setuptools.

In addition, there are three special prerelease tags that are treated as if they were the letter c: pre, preview, and rc. So, version 2.4rc1, 2.4pre1 and 2.4preview1 are all the exact same version as 2.4c1, and are treated as identical by setuptools.

A post-release tag is either a series of letters that are alphabetically greater than or equal to “final”, or a dash (-). Post-release tags are generally used to separate patch numbers, port numbers, build numbers, revision numbers, or date stamps from the release number. For example, the version 2.4-r1263 might denote Subversion revision 1263 of a post-release patch of version 2.4. Or you might use 2.4-20051127 to denote a date-stamped post-release.

Notice that after each pre or post-release tag, you are free to place another release number, followed again by more pre- or post-release tags. For example, 0.6a9.dev-r41475 could denote Subversion revision 41475 of the in-development version of the ninth alpha of release 0.6. Notice that dev is a pre-release tag, so this version is a *lower* version number than 0.6a9, which would be the actual ninth alpha of release 0.6. But the -r41475 is a post-release tag, so this version is *newer* than 0.6a9.dev.

For the most part, setuptools’ interpretation of version numbers is intuitive, but here are a few tips that will keep you out of trouble in the corner cases:

- Don’t stick adjoining pre-release tags together without a dot or number between them. Version 1.9adev is the adev prerelease of 1.9, *not* a development pre-release of 1.9a. Use .dev instead, as in 1.9a.dev, or separate the prerelease tags with a number, as in 1.9a0dev. 1.9a.dev, 1.9a0dev, and even 1.9.a.dev are identical versions from setuptools’ point of view, so you can use whatever scheme you prefer.
- If you want to be certain that your chosen numbering scheme works the way you think it will, you can use the `pkg_resources.parse_version()` function to compare different version numbers:

```
>>> from pkg_resources import parse_version
>>> parse_version("1.9.a.dev") == parse_version("1.9a0dev")
True
>>> parse_version("2.1-rc2") < parse_version("2.1")
True
>>> parse_version("0.6a9dev-r41475") < parse_version("0.6a9")
True
```

Once you’ve decided on a version numbering scheme for your project, you can have setuptools automatically tag your in-development releases with various pre- or post-release tags. See the following sections for more details:

- *Tagging and “Daily Build” or “Snapshot” Releases*
- The `egg_info` command

### 1.1.3 New and Changed `setup()` Keywords

The following keyword arguments to `setup()` are added or changed by setuptools. All of them are optional; you do not have to supply them unless you need the associated setuptools feature.

**include\_package\_data** If set to `True`, this tells setuptools to automatically include any data files it finds inside your package directories that are specified by your `MANIFEST.in` file. For more information, see the section below on *Including Data Files*.

**exclude\_package\_data** A dictionary mapping package names to lists of glob patterns that should be *excluded* from your package directories. You can use this to trim back any excess files included by `include_package_data`. For a complete description and examples, see the section below on *Including Data Files*.

**package\_data** A dictionary mapping package names to lists of glob patterns. For a complete description and examples, see the section below on *Including Data Files*. You do not need to use this option if you are using `include_package_data`, unless you need to add e.g. files that are generated by your setup script and build process. (And are therefore not in source control or are files that you don’t want to include in your source distribution.)

**zip\_safe** A boolean (`True` or `False`) flag specifying whether the project can be safely installed and run from a zip file. If this argument is not supplied, the `bdist_egg` command will have to analyze all of your project’s contents for possible problems each time it builds an egg.

**install\_requires** A string or list of strings specifying what other distributions need to be installed when this one is. See the section below on *Declaring Dependencies* for details and examples of the format of this argument.

**entry\_points** A dictionary mapping entry point group names to strings or lists of strings defining the entry points. Entry points are used to support dynamic discovery of services or plugins provided by a project. See *Dynamic Discovery of Services and Plugins* for details and examples of the format of this argument. In addition, this keyword is used to support *Automatic Script Creation*.

**extras\_require** A dictionary mapping names of “extras” (optional features of your project) to strings or lists of strings specifying what other distributions must be installed to support those features. See the section below on *Declaring Dependencies* for details and examples of the format of this argument.

**python\_requires** A string corresponding to a version specifier (as defined in PEP 440) for the Python version, used to specify the Requires-Python defined in PEP 345.

**setup\_requires** A string or list of strings specifying what other distributions need to be present in order for the *setup script* to run. setuptools will attempt to obtain these (using pip if available) before processing the rest of the setup script or commands. This argument is needed if you are using distutils extensions as part of your build process; for example, extensions that process `setup()` arguments and turn them into EGG-INFO metadata files.

(Note: projects listed in `setup_requires` will NOT be automatically installed on the system where the setup script is being run. They are simply downloaded to the `./eggs` directory if they’re not locally available already. If you want them to be installed, as well as being available when the setup script is run, you should add them to `install_requires` **and** `setup_requires`.)

**dependency\_links** A list of strings naming URLs to be searched when satisfying dependencies. These links will be used if needed to install packages specified by `setup_requires` or `tests_require`. They will also be written into the egg’s metadata for use during install by tools that support them.

**namespace\_packages** A list of strings naming the project’s “namespace packages”. A namespace package is a package that may be split across multiple project distributions. For example, Zope 3’s `zope` package is a namespace package, because subpackages like `zope.interface` and `zope.publisher` may be distributed separately. The egg runtime system can automatically merge such subpackages into a single parent package at runtime, as long as you declare them in each project that contains any subpackages of the namespace package, and as long as the namespace package’s `__init__.py` does not contain any code other than a namespace declaration. See the section below on *Namespace Packages* for more information.

**test\_suite** A string naming a `unittest.TestCase` subclass (or a package or module containing one or more of them, or a method of such a subclass), or naming a function that can be called with no arguments and returns a `unittest.TestSuite`. If the named suite is a module, and the module has an `additional_tests()` function, it is called and the results are added to the tests to be run. If the named suite is a package, any submodules and subpackages are recursively added to the overall test suite.

Specifying this argument enables use of the `test` command to run the specified test suite, e.g. via `setup.py test`. See the section on the `test` command below for more details.

New in 41.5.0: Deprecated the test command.

**tests\_require** If your project’s tests need one or more additional packages besides those needed to install it, you can use this option to specify them. It should be a string or list of strings specifying what other distributions need to be present for the package’s tests to run. When you run the `test` command, `setuptools` will attempt to obtain these (using `pip` if available). Note that these required projects will *not* be installed on the system where the tests are run, but only downloaded to the project’s setup directory if they’re not already installed locally.

New in 41.5.0: Deprecated the test command.

**test\_loader** If you would like to use a different way of finding tests to run than what `setuptools` normally uses, you can specify a module name and class name in this argument. The named class must be instantiable with no arguments, and its instances must support the `loadTestsFromNames()` method as defined in the Python `unittest` module’s `TestLoader` class. `Setuptools` will pass only one test “name” in the `names` argument: the value supplied for the `test_suite` argument. The loader you specify may interpret this string in any way it likes, as there are no restrictions on what may be contained in a `test_suite` string.

The module name and class name must be separated by a `:`. The default value of this argument is `"setuptools.command.test:ScanningLoader"`. If you want to use the default `unittest` behavior, you can specify `"unittest:TestLoader"` as your `test_loader` argument instead. This will prevent automatic scanning of submodules and subpackages.

The module and class you specify here may be contained in another package, as long as you use the `tests_require` option to ensure that the package containing the loader class is available when the `test` command is run.

New in 41.5.0: Deprecated the test command.

**eager\_resources** A list of strings naming resources that should be extracted together, if any of them is needed, or if any C extensions included in the project are imported. This argument is only useful if the project will be installed as a zipfile, and there is a need to have all of the listed resources be extracted to the filesystem *as a unit*. Resources listed here should be “/”-separated paths, relative to the source root, so to list a resource `foo.png` in package `bar.baz`, you would include the string `bar/baz/foo.png` in this argument.

If you only need to obtain resources one at a time, or you don’t have any C extensions that access other files in the project (such as data files or shared libraries), you probably do NOT need this argument and shouldn’t mess with it. For more details on how this argument works, see the section below on *Automatic Resource Extraction*.

**use\_2to3** Convert the source code from Python 2 to Python 3 with 2to3 during the build process. See *Supporting both Python 2 and Python 3 with Setuptools* for more details.

**convert\_2to3\_doctests** List of doctest source files that need to be converted with 2to3. See *Supporting both Python 2 and Python 3 with Setuptools* for more details.

**use\_2to3\_fixers** A list of modules to search for additional fixers to be used during the 2to3 conversion. See *Supporting both Python 2 and Python 3 with Setuptools* for more details.

**project\_urls** An arbitrary map of URL names to hyperlinks, allowing more extensible documentation of where various resources can be found than the simple `url` and `download_url` options provide.

### Using `find_packages()`

For simple projects, it's usually easy enough to manually add packages to the `packages` argument of `setup()`. However, for very large projects (Twisted, PEAK, Zope, Chandler, etc.), it can be a big burden to keep the package list updated. That's what `setuptools.find_packages()` is for.

`find_packages()` takes a source directory and two lists of package name patterns to exclude and include. If omitted, the source directory defaults to the same directory as the setup script. Some projects use a `src` or `lib` directory as the root of their source tree, and those projects would of course use `"src"` or `"lib"` as the first argument to `find_packages()`. (And such projects also need something like `package_dir={"": "src"}` in their `setup()` arguments, but that's just a normal distutils thing.)

Anyway, `find_packages()` walks the target directory, filtering by inclusion patterns, and finds Python packages (any directory). Packages are only recognized if they include an `__init__.py` file. Finally, exclusion patterns are applied to remove matching packages.

Inclusion and exclusion patterns are package names, optionally including wildcards. For example, `find_packages(exclude=["*.tests"])` will exclude all packages whose last name part is `tests`. Or, `find_packages(exclude=["*.tests", "*.tests.*"])` will also exclude any subpackages of packages named `tests`, but it still won't exclude a top-level `tests` package or the children thereof. In fact, if you really want no `tests` packages at all, you'll need something like this:

```
find_packages(exclude=["*.tests", "*.tests.*", "tests.*", "tests"])
```

in order to cover all the bases. Really, the exclusion patterns are intended to cover simpler use cases than this, like excluding a single, specified package and its subpackages.

Regardless of the parameters, the `find_packages()` function returns a list of package names suitable for use as the `packages` argument to `setup()`, and so is usually the easiest way to set that argument in your setup script. Especially since it frees you from having to remember to modify your setup script whenever your project grows additional top-level packages or subpackages.

### `find_namespace_packages()`

In Python 3.3+, `setuptools` also provides the `find_namespace_packages` variant of `find_packages`, which has the same function signature as `find_packages`, but works with [PEP 420](#) compliant implicit namespace packages. Here is a minimal setup script using `find_namespace_packages`:

```
from setuptools import setup, find_namespace_packages
setup(
    name="HelloWorld",
    version="0.1",
    packages=find_namespace_packages(),
)
```

Keep in mind that according to [PEP 420](#), you may have to either re-organize your codebase a bit or define a few exclusions, as the definition of an implicit namespace package is quite lenient, so for a project organized like so:

```

├── namespace
│   └── mypackage
│       ├── __init__.py
│       └── mod1.py
├── setup.py
├── tests
│   └── test_mod1.py

```

A naive `find_namespace_packages()` would install both `namespace.mypackage` and a top-level package called `tests`! One way to avoid this problem is to use the `include` keyword to whitelist the packages to include, like so:

```

from setuptools import setup, find_namespace_packages

setup(
    name="namespace.mypackage",
    version="0.1",
    packages=find_namespace_packages(include=["namespace.*"])
)

```

Another option is to use the “src” layout, where all package code is placed in the `src` directory, like so:

```

├── setup.py
├── src
│   ├── namespace
│   │   └── mypackage
│   │       ├── __init__.py
│   │       └── mod1.py
├── tests
│   └── test_mod1.py

```

With this layout, the package directory is specified as `src`, as such:

```

setup(name="namespace.mypackage",
      version="0.1",
      package_dir={"": "src"},
      packages=find_namespace_packages(where="src"))

```

## 1.1.4 Automatic Script Creation

Packaging and installing scripts can be a bit awkward with the distutils. For one thing, there’s no easy way to have a script’s filename match local conventions on both Windows and POSIX platforms. For another, you often have to create a separate file just for the “main” script, when your actual “main” is a function in a module somewhere. And even in Python 2.4, using the `-m` option only works for actual `.py` files that aren’t installed in a package.

`setuptools` fixes all of these problems by automatically generating scripts for you with the correct extension, and on Windows it will even create an `.exe` file so that users don’t have to change their `PATHEXT` settings. The way to use this feature is to define “entry points” in your setup script that indicate what function the generated script should import and run. For example, to create two console scripts called `foo` and `bar`, and a GUI script called `baz`, you might do something like this:

```

setup(
    # other arguments here...
    entry_points={
        "console_scripts": [

```

(continues on next page)



(continued from previous page)

```
        "foo = my_package.some_module:main_func",
        "bar = other_module:some_func",
    ],
    "gui_scripts": [
        "baz = my_package_gui:start_func",
    ]
}
)
```

When this project is installed on non-Windows platforms (using “`setup.py install`”, “`setup.py develop`”, or with `pip`), a set of `foo`, `bar`, and `baz` scripts will be installed that import `main_func` and `some_func` from the specified modules. The functions you specify are called with no arguments, and their return value is passed to `sys.exit()`, so you can return an errorlevel or message to print to `stderr`.

On Windows, a set of `foo.exe`, `bar.exe`, and `baz.exe` launchers are created, alongside a set of `foo.py`, `bar.py`, and `baz.pyw` files. The `.exe` wrappers find and execute the right version of Python to run the `.py` or `.pyw` file.

You may define as many “console script” and “gui script” entry points as you like, and each one can optionally specify “extras” that it depends on, that will be added to `sys.path` when the script is run. For more information on “extras”, see the section below on *Declaring Extras*. For more information on “entry points” in general, see the section below on *Dynamic Discovery of Services and Plugins*.

## “Eggsecutable” Scripts

Deprecated since version 45.3.0.

Occasionally, there are situations where it’s desirable to make an `.egg` file directly executable. You can do this by including an entry point such as the following:

```
setup(
    # other arguments here...
    entry_points={
        "setuptools.installation": [
            "eggsecutable = my_package.some_module:main_func",
        ]
    }
)
```

Any eggs built from the above setup script will include a short executable prelude that imports and calls `main_func()` from `my_package.some_module`. The prelude can be run on Unix-like platforms (including Mac and Linux) by invoking the egg with `/bin/sh`, or by enabling execute permissions on the `.egg` file. For the executable prelude to run, the appropriate version of Python must be available via the `PATH` environment variable, under its “long” name. That is, if the egg is built for Python 2.3, there must be a `python2.3` executable present in a directory on `PATH`.

**IMPORTANT NOTE:** Eggs with an “eggsecutable” header cannot be renamed, or invoked via symlinks. They *must* be invoked using their original filename, in order to ensure that, once running, `pkg_resources` will know what project and version is in use. The header script will check this and exit with an error if the `.egg` file has been renamed or is invoked via a symlink that changes its base name.

## 1.1.5 Declaring Dependencies

`setuptools` supports automatically installing dependencies when a package is installed, and including information about dependencies in Python Eggs (so that package management tools like `pip` can use the information).



setuptools and pkg\_resources use a common syntax for specifying a project's required dependencies. This syntax consists of a project's PyPI name, optionally followed by a comma-separated list of "extras" in square brackets, optionally followed by a comma-separated list of version specifiers. A version specifier is one of the operators `<`, `>`, `<=`, `>=`, `==` or `!=`, followed by a version identifier. Tokens may be separated by whitespace, but any whitespace or nonstandard characters within a project name or version identifier must be replaced with `-`.

Version specifiers for a given project are internally sorted into ascending version order, and used to establish what ranges of versions are acceptable. Adjacent redundant conditions are also consolidated (e.g. `">1, >2"` becomes `">2"`, and `"<2, <3"` becomes `"<2"`). `"!="` versions are excised from the ranges they fall within. A project's version is then checked for membership in the resulting ranges. (Note that providing conflicting conditions for the same version (e.g. `"<2,>=2"` or `"==2,!2"`) is meaningless and may therefore produce bizarre results.)

Here are some example requirement specifiers:

```
docutils >= 0.3

# comment lines and \ continuations are allowed in requirement strings
BazSpam ==1.1, ==1.2, ==1.3, ==1.4, ==1.5, \
    ==1.6, ==1.7 # and so are line-end comments

PEAK[FastCGI, reST]>=0.5a4

setuptools==0.5a7
```

The simplest way to include requirement specifiers is to use the `install_requires` argument to `setup()`. It takes a string or list of strings containing requirement specifiers. If you include more than one requirement in a string, each requirement must begin on a new line.

This has three effects:

1. When your project is installed, either by using `pip`, `setup.py install`, or `setup.py develop`, all of the dependencies not already installed will be located (via PyPI), downloaded, built (if necessary), and installed.
2. Any scripts in your project will be installed with wrappers that verify the availability of the specified dependencies at runtime, and ensure that the correct versions are added to `sys.path` (e.g. if multiple versions have been installed).
3. Python Egg distributions will include a metadata file listing the dependencies.

Note, by the way, that if you declare your dependencies in `setup.py`, you do *not* need to use the `require()` function in your scripts or modules, as long as you either install the project or use `setup.py develop` to do development work on it. (See *"Development Mode"* below for more details on using `setup.py develop`.)

## Dependencies that aren't in PyPI

**Warning:** Dependency links support has been dropped by pip starting with version 19.0 (released 2019-01-22).

If your project depends on packages that don't exist on PyPI, you may still be able to depend on them, as long as they are available for download as:

- an egg, in the standard distutils `sdist` format,
- a single `.py` file, or
- a VCS repository (Subversion, Mercurial, or Git).

You just need to add some URLs to the `dependency_links` argument to `setup()`.

The URLs must be either:

1. direct download URLs,
2. the URLs of web pages that contain direct download links, or
3. the repository's URL

In general, it's better to link to web pages, because it is usually less complex to update a web page than to release a new version of your project. You can also use a SourceForge `showfiles.php` link in the case where a package you depend on is distributed via SourceForge.

If you depend on a package that's distributed as a single `.py` file, you must include an `"#egg=project-version"` suffix to the URL, to give a project name and version number. (Be sure to escape any dashes in the name or version by replacing them with underscores.) EasyInstall will recognize this suffix and automatically create a trivial `setup.py` to wrap the single `.py` file as an egg.

In the case of a VCS checkout, you should also append `#egg=project-version` in order to identify for what package that checkout should be used. You can append `@REV` to the URL's path (before the fragment) to specify a revision. Additionally, you can also force the VCS being used by prepending the URL with a certain prefix. Currently available are:

- `svn+URL` for Subversion,
- `git+URL` for Git, and
- `hg+URL` for Mercurial

A more complete example would be:

```
vcs+proto://host/path@revision#egg=project-version
```

Be careful with the version. It should match the one inside the project files. If you want to disregard the version, you have to omit it both in the `requires` and in the URL's fragment.

This will do a checkout (or a clone, in Git and Mercurial parlance) to a temporary folder and run `setup.py bdist_egg`.

The `dependency_links` option takes the form of a list of URL strings. For example, this will cause a search of the specified page for eggs or source distributions, if the package's dependencies aren't already installed:

```
setup(
    ...
    dependency_links=[
        "http://peak.telecommunity.com/snapshots/"
    ],
)
```

## Declaring “Extras” (optional features with their own dependencies)

Sometimes a project has “recommended” dependencies, that are not required for all uses of the project. For example, a project might offer optional PDF output if ReportLab is installed, and reStructuredText support if docutils is installed. These optional features are called “extras”, and setuptools allows you to define their requirements as well. In this way, other projects that require these optional features can force the additional requirements to be installed, by naming the desired extras in their `install_requires`.

For example, let's say that Project A offers optional PDF and reST support:

```
setup(
    name="Project-A",
    ...
```

(continues on next page)

(continued from previous page)

```

extras_require={
    "PDF": ["ReportLab>=1.2", "RXP"],
    "reST": ["docutils>=0.3"],
}
)

```

As you can see, the `extras_require` argument takes a dictionary mapping names of “extra” features, to strings or lists of strings describing those features’ requirements. These requirements will *not* be automatically installed unless another package depends on them (directly or indirectly) by including the desired “extras” in square brackets after the associated project name. (Or if the extras were listed in a requirement spec on the “pip install” command line.)

Extras can be used by a project’s *entry points* to specify dynamic dependencies. For example, if Project A includes a “rst2pdf” script, it might declare it like this, so that the “PDF” requirements are only resolved if the “rst2pdf” script is run:

```

setup(
    name="Project-A",
    ...
    entry_points={
        "console_scripts": [
            "rst2pdf = project_a.tools.pdfgen [PDF]",
            "rst2html = project_a.tools.htmlgen",
            # more script entry points ...
        ],
    }
)

```

Projects can also use another project’s extras when specifying dependencies. For example, if project B needs “project A” with PDF support installed, it might declare the dependency like this:

```

setup(
    name="Project-B",
    install_requires=["Project-A[PDF]"],
    ...
)

```

This will cause ReportLab to be installed along with project A, if project B is installed – even if project A was already installed. In this way, a project can encapsulate groups of optional “downstream dependencies” under a feature name, so that packages that depend on it don’t have to know what the downstream dependencies are. If a later version of Project A builds in PDF support and no longer needs ReportLab, or if it ends up needing other dependencies besides ReportLab in order to provide PDF support, Project B’s setup information does not need to change, but the right packages will still be installed if needed.

Note, by the way, that if a project ends up not needing any other packages to support a feature, it should keep an empty requirements list for that feature in its `extras_require` argument, so that packages depending on that feature don’t break (due to an invalid feature name). For example, if Project A above builds in PDF support and no longer needs ReportLab, it could change its setup to this:

```

setup(
    name="Project-A",
    ...
    extras_require={
        "PDF": [],
        "reST": ["docutils>=0.3"],
    }
)

```

so that Package B doesn't have to remove the [PDF] from its requirement specifier.

## Declaring platform specific dependencies

Sometimes a project might require a dependency to run on a specific platform. This could be a package that back ports a module so that it can be used in older python versions. Or it could be a package that is required to run on a specific operating system. This will allow a project to work on multiple different platforms without installing dependencies that are not required for a platform that is installing the project.

For example, here is a project that uses the `enum` module and `pywin32`:

```
setup(
    name="Project",
    ...
    install_requires=[
        "enum34;python_version<'3.4'",
        "pywin32 >= 1.0;platform_system=='Windows'"
    ]
)
```

Since the `enum` module was added in Python 3.4, it should only be installed if the python version is earlier. Since `pywin32` will only be used on windows, it should only be installed when the operating system is Windows. Specifying version requirements for the dependencies is supported as normal.

The environmental markers that may be used for testing platform types are detailed in [PEP 508](#).

### 1.1.6 Including Data Files

The distutils have traditionally allowed installation of “data files”, which are placed in a platform-specific location. However, the most common use case for data files distributed with a package is for use *by* the package, usually by including the data files in the package directory.

Setuptools offers three ways to specify data files to be included in your packages. First, you can simply use the `include_package_data` keyword, e.g.:

```
from setuptools import setup, find_packages
setup(
    ...
    include_package_data=True
)
```

This tells setuptools to install any data files it finds in your packages. The data files must be specified via the distutils' `MANIFEST.in` file. (They can also be tracked by a revision control system, using an appropriate plugin. See the section below on [Adding Support for Revision Control Systems](#) for information on how to write such plugins.)

If you want finer-grained control over what files are included (for example, if you have documentation files in your package directories and want to exclude them from installation), then you can also use the `package_data` keyword, e.g.:

```
from setuptools import setup, find_packages
setup(
    ...
    package_data={
        # If any package contains *.txt or *.rst files, include them:
        "": ["*.txt", "*.rst"],
        # And include any *.msg files found in the "hello" package, too:
    }
```

(continues on next page)

(continued from previous page)

```

        "hello": ["*.msg"],
    }
)

```

The `package_data` argument is a dictionary that maps from package names to lists of glob patterns. The globs may include subdirectory names, if the data files are contained in a subdirectory of the package. For example, if the package tree looks like this:

```

setup.py
src/
    mypkg/
        __init__.py
        mypkg.txt
        data/
            somefile.dat
            otherdata.dat

```

The setuptools setup file might look like this:

```

from setuptools import setup, find_packages
setup(
    ...
    packages=find_packages("src"), # include all packages under src
    package_dir={"": "src"},      # tell distutils packages are under src

    package_data={
        # If any package contains *.txt files, include them:
        "": ["*.txt"],
        # And include any *.dat files found in the "data" subdirectory
        # of the "mypkg" package, also:
        "mypkg": ["data/*.dat"],
    }
)

```

Notice that if you list patterns in `package_data` under the empty string, these patterns are used to find files in every package, even ones that also have their own patterns listed. Thus, in the above example, the `mypkg.txt` file gets included even though it's not listed in the patterns for `mypkg`.

Also notice that if you use paths, you *must* use a forward slash (/) as the path separator, even if you are on Windows. Setuptools automatically converts slashes to appropriate platform-specific separators at build time.

If datafiles are contained in a subdirectory of a package that isn't a package itself (no `__init__.py`), then the subdirectory names (or `*`) are required in the `package_data` argument (as shown above with `"data/*.dat"`).

When building an sdist, the datafiles are also drawn from the `package_name.egg-info/SOURCES.txt` file, so make sure that this is removed if the `setup.py package_data` list is updated before calling `setup.py`.

(Note: although the `package_data` argument was previously only available in setuptools, it was also added to the Python distutils package as of Python 2.4; there is [some documentation for the feature](#) available on the python.org website. If using the setuptools-specific `include_package_data` argument, files specified by `package_data` will *not* be automatically added to the manifest unless they are listed in the MANIFEST.in file.)

Sometimes, the `include_package_data` or `package_data` options alone aren't sufficient to precisely define what files you want included. For example, you may want to include package README files in your revision control system and source distributions, but exclude them from being installed. So, setuptools offers an `exclude_package_data` option as well, that allows you to do things like this:

```
from setuptools import setup, find_packages
setup(
    ...
    packages=find_packages("src"), # include all packages under src
    package_dir={"": "src"},      # tell distutils packages are under src

    include_package_data=True,     # include everything in source control

    # ...but exclude README.txt from all packages
    exclude_package_data={"": ["README.txt"]},
)
```

The `exclude_package_data` option is a dictionary mapping package names to lists of wildcard patterns, just like the `package_data` option. And, just as with that option, a key of `"` will apply the given pattern(s) to all packages. However, any files that match these patterns will be *excluded* from installation, even if they were listed in `package_data` or were included as a result of using `include_package_data`.

In summary, the three options allow you to:

**include\_package\_data** Accept all data files and directories matched by `MANIFEST.in`.

**package\_data** Specify additional patterns to match files that may or may not be matched by `MANIFEST.in` or found in source control.

**exclude\_package\_data** Specify patterns for data files and directories that should *not* be included when a package is installed, even if they would otherwise have been included due to the use of the preceding options.

NOTE: Due to the way the distutils build process works, a data file that you include in your project and then stop including may be “orphaned” in your project’s build directories, requiring you to run `setup.py clean --all` to fully remove them. This may also be important for your users and contributors if they track intermediate revisions of your project using Subversion; be sure to let them know when you make changes that remove files from inclusion so they can run `setup.py clean --all`.

## Accessing Data Files at Runtime

Typically, existing programs manipulate a package’s `__file__` attribute in order to find the location of data files. However, this manipulation isn’t compatible with PEP 302-based import hooks, including importing from zip files and Python Eggs. It is strongly recommended that, if you are using data files, you should use the [ResourceManager API](#) of `pkg_resources` to access them. The `pkg_resources` module is distributed as part of setuptools, so if you’re using setuptools to distribute your package, there is no reason not to use its resource management API. See also [Importlib Resources](#) for a quick example of converting code that uses `__file__` to use `pkg_resources` instead.

## Non-Package Data Files

Historically, setuptools by way of `easy_install` would encapsulate data files from the distribution into the egg (see [the old docs](#)). As eggs are deprecated and pip-based installs fall back to the platform-specific location for installing data files, there is no supported facility to reliably retrieve these resources.

Instead, the PyPA recommends that any data files you wish to be accessible at run time be included in the package.

## Automatic Resource Extraction

If you are using tools that expect your resources to be “real” files, or your project includes non-extension native libraries or other files that your C extensions expect to be able to access, you may need to list those files in the

`eager_resources` argument to `setup()`, so that the files will be extracted together, whenever a C extension in the project is imported.

This is especially important if your project includes shared libraries *other* than distutils-built C extensions, and those shared libraries use file extensions other than `.dll`, `.so`, or `.dylib`, which are the extensions that setuptools 0.6a8 and higher automatically detects as shared libraries and adds to the `native_libs.txt` file for you. Any shared libraries whose names do not end with one of those extensions should be listed as `eager_resources`, because they need to be present in the filesystem when the C extensions that link to them are used.

The `pkg_resources` runtime for compressed packages will automatically extract *all* C extensions and `eager_resources` at the same time, whenever *any* C extension or eager resource is requested via the `resource_filename()` API. (C extensions are imported using `resource_filename()` internally.) This ensures that C extensions will see all of the “real” files that they expect to see.

Note also that you can list directory resource names in `eager_resources` as well, in which case the directory’s contents (including subdirectories) will be extracted whenever any C extension or eager resource is requested.

Please note that if you’re not sure whether you need to use this argument, you don’t! It’s really intended to support projects with lots of non-Python dependencies and as a last resort for cruffy projects that can’t otherwise handle being compressed. If your package is pure Python, Python plus data files, or Python plus C, you really don’t need this. You’ve got to be using either C or an external program that needs “real” files in your project before there’s any possibility of `eager_resources` being relevant to your project.

## 1.1.7 Extensible Applications and Frameworks

### Dynamic Discovery of Services and Plugins

setuptools supports creating libraries that “plug in” to extensible applications and frameworks, by letting you register “entry points” in your project that can be imported by the application or framework.

For example, suppose that a blogging tool wants to support plugins that provide translation for various file types to the blog’s output format. The framework might define an “entry point group” called `blogtool.parsers`, and then allow plugins to register entry points for the file extensions they support.

This would allow people to create distributions that contain one or more parsers for different file types, and then the blogging tool would be able to find the parsers at runtime by looking up an entry point for the file extension (or mime type, or however it wants to).

Note that if the blogging tool includes parsers for certain file formats, it can register these as entry points in its own setup script, which means it doesn’t have to special-case its built-in formats. They can just be treated the same as any other plugin’s entry points would be.

If you’re creating a project that plugs in to an existing application or framework, you’ll need to know what entry points or entry point groups are defined by that application or framework. Then, you can register entry points in your setup script. Here are a few examples of ways you might register an `.rst` file parser entry point in the `blogtool.parsers` entry point group, for our hypothetical blogging tool:

```
setup(
    # ...
    entry_points={"blogtool.parsers": ".rst = some_module:SomeClass"}
)

setup(
    # ...
    entry_points={"blogtool.parsers": [".rst = some_module:a_func"]}
```

(continues on next page)

(continued from previous page)

```
setup(  
    # ...  
    entry_points="""  
        [blogtool.parsers]  
        .rst = some.nested.module:SomeClass.some_classmethod [reST]  
    """,  
    extras_require=dict(reST="Docutils>=0.3.5")  
)
```

The `entry_points` argument to `setup()` accepts either a string with `.ini`-style sections, or a dictionary mapping entry point group names to either strings or lists of strings containing entry point specifiers. An entry point specifier consists of a name and value, separated by an `=` sign. The value consists of a dotted module name, optionally followed by a `:` and a dotted identifier naming an object within the module. It can also include a bracketed list of “extras” that are required for the entry point to be used. When the invoking application or framework requests loading of an entry point, any requirements implied by the associated extras will be passed to `pkg_resources.require()`, so that an appropriate error message can be displayed if the needed package(s) are missing. (Of course, the invoking app or framework can ignore such errors if it wants to make an entry point optional if a requirement isn’t installed.)

## Defining Additional Metadata

Some extensible applications and frameworks may need to define their own kinds of metadata to include in eggs, which they can then access using the `pkg_resources` metadata APIs. Ordinarily, this is done by having plugin developers include additional files in their `ProjectName.egg-info` directory. However, since it can be tedious to create such files by hand, you may want to create a `distutils` extension that will create the necessary files from arguments to `setup()`, in much the same way that `setuptools` does for many of the `setup()` arguments it adds. See the section below on *Creating distutils Extensions* for more details, especially the subsection on *Adding new EGG-INFO Files*.

### 1.1.8 “Development Mode”

Under normal circumstances, the `distutils` assume that you are going to build a distribution of your project, not use it in its “raw” or “unbuilt” form. If you were to use the `distutils` that way, you would have to rebuild and reinstall your project every time you made a change to it during development.

Another problem that sometimes comes up with the `distutils` is that you may need to do development on two related projects at the same time. You may need to put both projects’ packages in the same directory to run them, but need to keep them separate for revision control purposes. How can you do this?

Setuptools allows you to deploy your projects for use in a common directory or staging area, but without copying any files. Thus, you can edit each project’s code in its checkout directory, and only need to run build commands when you change a project’s C extensions or similarly compiled files. You can even deploy a project into another project’s checkout directory, if that’s your preferred way of working (as opposed to using a common independent staging area or the `site-packages` directory).

To do this, use the `setup.py develop` command. It works very similarly to `setup.py install`, except that it doesn’t actually install anything. Instead, it creates a special `.egg-link` file in the deployment directory, that links to your project’s source code. And, if your deployment directory is Python’s `site-packages` directory, it will also update the `easy-install.pth` file to include your project’s source code, thereby making it available on `sys.path` for all programs using that Python installation.

If you have enabled the `use_2to3` flag, then of course the `.egg-link` will not link directly to your source code when run under Python 3, since that source code would be made for Python 2 and not work under Python 3. Instead the `setup.py develop` will build Python 3 code under the `build` directory, and link there. This means that after



doing code changes you will have to run `setup.py build` before these changes are picked up by your Python 3 installation.

In addition, the `develop` command creates wrapper scripts in the target script directory that will run your in-development scripts after ensuring that all your `install_requires` packages are available on `sys.path`.

You can deploy the same project to multiple staging areas, e.g. if you have multiple projects on the same machine that are sharing the same project you're doing development work.

When you're done with a given development task, you can remove the project source from a staging area using `setup.py develop --uninstall`, specifying the desired staging area if it's not the default.

There are several options to control the precise behavior of the `develop` command; see the section on the [develop](#) command below for more details.

Note that you can also apply setuptools commands to non-setuptools projects, using commands like this:

```
python -c "import setuptools; with open('setup.py') as f: exec(compile(f.read(),
↪ 'setup.py', 'exec'))" develop
```

That is, you can simply list the normal setup commands and options following the quoted part.

### 1.1.9 Distributing a setuptools-based project

Detailed instructions to distribute a setuptools project can be found at [Packaging project tutorials](#).

Before you begin, make sure you have the latest versions of setuptools and wheel:

```
pip install --upgrade setuptools wheel
```

To build a setuptools project, run this command from the same directory where `setup.py` is located:

```
setup.py sdist bdist_wheel
```

This will generate distribution archives in the `dist` directory.

Before you upload the generated archives make sure you're registered on <https://test.pypi.org/account/register/>. You will also need to verify your email to be able to upload any packages. You should install twine to be able to upload packages:

```
pip install --upgrade twine
```

Now, to upload these archives, run:

```
twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

To install your newly uploaded package `example_pkg`, you can use `pip`:

```
pip install --index-url https://test.pypi.org/simple/ example_pkg
```

If you have issues at any point, please refer to [Packaging project tutorials](#) for clarification.

#### Setting the `zip_safe` flag

For some use cases (such as bundling as part of a larger application), Python packages may be run directly from a zip file. Not all packages, however, are capable of running in compressed form, because they may expect to be able to access either source code or data files as normal operating system files. So, setuptools can install your project as a zipfile or a directory, and its default choice is determined by the project's `zip_safe` flag.

You can pass a `True` or `False` value for the `zip_safe` argument to the `setup()` function, or you can omit it. If you omit it, the `bdist_egg` command will analyze your project's contents to see if it can detect any conditions that would prevent it from working in a zipfile. It will output notices to the console about any such conditions that it finds.

Currently, this analysis is extremely conservative: it will consider the project unsafe if it contains any C extensions or datafiles whatsoever. This does *not* mean that the project can't or won't work as a zipfile! It just means that the `bdist_egg` authors aren't yet comfortable asserting that the project *will* work. If the project contains no C or data files, and does no `__file__` or `__path__` introspection or source code manipulation, then there is an extremely solid chance the project will work when installed as a zipfile. (And if the project uses `pkg_resources` for all its data file access, then C extensions and other data files shouldn't be a problem at all. See the [Accessing Data Files at Runtime](#) section above for more information.)

However, if `bdist_egg` can't be *sure* that your package will work, but you've checked over all the warnings it issued, and you are either satisfied it *will* work (or if you want to try it for yourself), then you should set `zip_safe` to `True` in your `setup()` call. If it turns out that it doesn't work, you can always change it to `False`, which will force `setuptools` to install your project as a directory rather than as a zipfile.

In the future, as we gain more experience with different packages and become more satisfied with the robustness of the `pkg_resources` runtime, the “zip safety” analysis may become less conservative. However, we strongly recommend that you determine for yourself whether your project functions correctly when installed as a zipfile, correct any problems if you can, and then make an explicit declaration of `True` or `False` for the `zip_safe` flag, so that it will not be necessary for `bdist_egg` to try to guess whether your project can work as a zipfile.

## Namespace Packages

Sometimes, a large package is more useful if distributed as a collection of smaller eggs. However, Python does not normally allow the contents of a package to be retrieved from more than one location. “Namespace packages” are a solution for this problem. When you declare a package to be a namespace package, it means that the package has no meaningful contents in its `__init__.py`, and that it is merely a container for modules and subpackages.

The `pkg_resources` runtime will then automatically ensure that the contents of namespace packages that are spread over multiple eggs or directories are combined into a single “virtual” package.

The `namespace_packages` argument to `setup()` lets you declare your project's namespace packages, so that they will be included in your project's metadata. The argument should list the namespace packages that the egg participates in. For example, the `ZopeInterface` project might do this:

```
setup(
    # ...
    namespace_packages=["zope"]
)
```

because it contains a `zope.interface` package that lives in the `zope` namespace package. Similarly, a project for a standalone `zope.publisher` would also declare the `zope` namespace package. When these projects are installed and used, Python will see them both as part of a “virtual” `zope` package, even though they will be installed in different locations.

Namespace packages don't have to be top-level packages. For example, Zope 3's `zope.app` package is a namespace package, and in the future PEAK's `peak.util` package will be too.

Note, by the way, that your project's source tree must include the namespace packages' `__init__.py` files (and the `__init__.py` of any parent packages), in a normal Python package layout. These `__init__.py` files *must* contain the line:

```
__import__("pkg_resources").declare_namespace(__name__)
```

This code ensures that the namespace package machinery is operating and that the current package is registered as a namespace package.

You must NOT include any other code and data in a namespace package's `__init__.py`. Even though it may appear to work during development, or when projects are installed as `.egg` files, it will not work when the projects are installed using “system” packaging tools – in such cases the `__init__.py` files will not be installed, let alone executed.

You must include the `declare_namespace()` line in the `__init__.py` of *every* project that has contents for the namespace package in question, in order to ensure that the namespace will be declared regardless of which project's copy of `__init__.py` is loaded first. If the first loaded `__init__.py` doesn't declare it, it will never *be* declared, because no other copies will ever be loaded!

## TRANSITIONAL NOTE

Setuptools automatically calls `declare_namespace()` for you at runtime, but future versions may *not*. This is because the automatic declaration feature has some negative side effects, such as needing to import all namespace packages during the initialization of the `pkg_resources` runtime, and also the need for `pkg_resources` to be explicitly imported before any namespace packages work at all. In some future releases, you'll be responsible for including your own declaration lines, and the automatic declaration feature will be dropped to get rid of the negative side effects.

During the remainder of the current development cycle, therefore, setuptools will warn you about missing `declare_namespace()` calls in your `__init__.py` files, and you should correct these as soon as possible before the compatibility support is removed. Namespace packages without declaration lines will not work correctly once a user has upgraded to a later version, so it's important that you make this change now in order to avoid having your code break in the field. Our apologies for the inconvenience, and thank you for your patience.

## Tagging and “Daily Build” or “Snapshot” Releases

When a set of related projects are under development, it may be important to track finer-grained version increments than you would normally use for e.g. “stable” releases. While stable releases might be measured in dotted numbers with alpha/beta/etc. status codes, development versions of a project often need to be tracked by revision or build number or even build date. This is especially true when projects in development need to refer to one another, and therefore may literally need an up-to-the-minute version of something!

To support these scenarios, setuptools allows you to “tag” your source and egg distributions by adding one or more of the following to the project's “official” version identifier:

- A manually-specified pre-release tag, such as “build” or “dev”, or a manually-specified post-release tag, such as a build or revision number (`--tag-build=STRING`, `-bSTRING`)
- An 8-character representation of the build date (`--tag-date`, `-d`), as a postrelease tag

You can add these tags by adding `egg_info` and the desired options to the command line ahead of the `sdist` or `bdist` commands that you want to generate a daily build or snapshot for. See the section below on the `egg_info` command for more details.

(Also, before you release your project, be sure to see the section above on *Specifying Your Project's Version* for more information about how pre- and post-release tags affect how version numbers are interpreted. This is important in order to make sure that dependency processing tools will know which versions of your project are newer than others.)

Finally, if you are creating builds frequently, and either building them in a downloadable location or are copying them to a distribution server, you should probably also check out the `rotate` command, which lets you automatically delete all but the N most-recently-modified distributions matching a glob pattern. So, you can use a command line like:

```
setup.py egg_info -rbDEV bdist_egg rotate -m.egg -k3
```

to build an egg whose version info includes “DEV-rNNNN” (where NNNN is the most recent Subversion revision that affected the source tree), and then delete any egg files from the distribution directory except for the three that were built most recently.

If you have to manage automated builds for multiple packages, each with different tagging and rotation policies, you may also want to check out the *alias* command, which would let each package define an alias like `daily` that would perform the necessary tag, build, and rotate commands. Then, a simpler script or cron job could just run `setup.py daily` in each project directory. (And, you could also define sitewide or per-user default versions of the `daily` alias, so that projects that didn’t define their own would use the appropriate defaults.)

### Generating Source Distributions

`setuptools` enhances the `distutils`’ default algorithm for source file selection with pluggable endpoints for looking up files to include. If you are using a revision control system, and your source distributions only need to include files that you’re tracking in revision control, use a corresponding plugin instead of writing a `MANIFEST.in` file. See the section below on *Adding Support for Revision Control Systems* for information on plugins.

If you need to include automatically generated files, or files that are kept in an unsupported revision control system, you’ll need to create a `MANIFEST.in` file to specify any files that the default file location algorithm doesn’t catch. See the `distutils` documentation for more information on the format of the `MANIFEST.in` file.

But, be sure to ignore any part of the `distutils` documentation that deals with `MANIFEST` or how it’s generated from `MANIFEST.in`; `setuptools` shields you from these issues and doesn’t work the same way in any case. Unlike the `distutils`, `setuptools` regenerates the source distribution manifest file every time you build a source distribution, and it builds it inside the project’s `.egg-info` directory, out of the way of your main project directory. You therefore need not worry about whether it is up-to-date or not.

Indeed, because `setuptools`’ approach to determining the contents of a source distribution is so much simpler, its `sdist` command omits nearly all of the options that the `distutils`’ more complex `sdist` process requires. For all practical purposes, you’ll probably use only the `--formats` option, if you use any option at all.

### Making “Official” (Non-Snapshot) Releases

When you make an official release, creating source or binary distributions, you will need to override the tag settings from `setup.cfg`, so that you don’t end up registering versions like `foobar-0.7a1.dev-r34832`. This is easy to do if you are developing on the trunk and using tags or branches for your releases - just make the change to `setup.cfg` after branching or tagging the release, so the trunk will still produce development snapshots.

Alternately, if you are not branching for releases, you can override the default version options on the command line, using something like:

```
setup.py egg_info -Db "" sdist bdist_egg
```

The first part of this command (`egg_info -Db ""`) will override the configured tag information, before creating source and binary eggs. Thus, these commands will use the plain version from your `setup.py`, without adding the build designation string.

Of course, if you will be doing this a lot, you may wish to create a personal alias for this operation, e.g.:

```
setup.py alias -u release egg_info -Db ""
```

You can then use it like this:

```
setup.py release sdist bdist_egg
```

Or of course you can create more elaborate aliases that do all of the above. See the sections below on the *egg\_info* and *alias* commands for more ideas.

## Distributing Extensions compiled with Cython

setuptools will detect at build time whether Cython is installed or not. If Cython is not found setuptools will ignore pyx files.

To ensure Cython is available, include Cython in the build-requires section of your pyproject.toml:

```
[build-system]
requires=[..., "cython"]
```

Built with pip 10 or later, that declaration is sufficient to include Cython in the build. For broader compatibility, declare the dependency in your setup-requires of setup.cfg:

```
[options]
setup_requires =
    ...
    cython
```

As long as Cython is present in the build environment, setuptools includes transparent support for building Cython extensions, as long as extensions are defined using `setuptools.Extension`.

If you follow these rules, you can safely list `.pyx` files as the source of your `Extension` objects in the setup script. If it is, then setuptools will use it.

Of course, for this to work, your source distributions must include the C code generated by Cython, as well as your original `.pyx` files. This means that you will probably want to include current `.c` files in your revision control system, rebuilding them whenever you check changes in for the `.pyx` source files. This will ensure that people tracking your project in a revision control system will be able to build it even if they don't have Cython installed, and that your source releases will be similarly usable with or without Cython.

## 1.2 Command Reference

### 1.2.1 *alias* - Define shortcuts for commonly used commands

Sometimes, you need to use the same commands over and over, but you can't necessarily set them as defaults. For example, if you produce both development snapshot releases and "stable" releases of a project, you may want to put the distributions in different places, or use different *egg\_info* tagging options, etc. In these cases, it doesn't make sense to set the options in a distutils configuration file, because the values of the options changed based on what you're trying to do.

Setuptools therefore allows you to define "aliases" - shortcut names for an arbitrary string of commands and options, using `setup.py alias aliasname expansion`, where *aliasname* is the name of the new alias, and the remainder of the command line supplies its expansion. For example, this command defines a sitewide alias called "daily", that sets various *egg\_info* tagging options:

```
setup.py alias --global-config daily egg_info --tag-build=development
```

Once the alias is defined, it can then be used with other setup commands, e.g.:

```
setup.py daily bdist_egg      # generate a daily-build .egg file
setup.py daily sdist         # generate a daily-build source distro
setup.py daily sdist bdist_egg # generate both
```

The above commands are interpreted as if the word `daily` were replaced with `egg_info --tag-build=development`.

Note that setuptools will expand each alias *at most once* in a given command line. This serves two purposes. First, if you accidentally create an alias loop, it will have no effect; you'll instead get an error message about an unknown command. Second, it allows you to define an alias for a command, that uses that command. For example, this (project-local) alias:

```
setup.py alias bdist_egg bdist_egg rotate -kl -m.egg
```

redefines the `bdist_egg` command so that it always runs the `rotate` command afterwards to delete all but the newest egg file. It doesn't loop indefinitely on `bdist_egg` because the alias is only expanded once when used.

You can remove a defined alias with the `--remove` (or `-r`) option, e.g.:

```
setup.py alias --global-config --remove daily
```

would delete the “daily” alias we defined above.

Aliases can be defined on a project-specific, per-user, or sitewide basis. The default is to define or remove a project-specific alias, but you can use any of the [configuration file options](#) (listed under the [saveopts](#) command, below) to determine which distutils configuration file an aliases will be added to (or removed from).

Note that if you omit the “expansion” argument to the `alias` command, you'll get output showing that alias' current definition (and what configuration file it's defined in). If you omit the alias name as well, you'll get a listing of all current aliases along with their configuration file locations.

## 1.2.2 `bdist_egg` - Create a Python Egg for the project

**Warning:** `eggs` are deprecated in favor of wheels, and not supported by pip.

This command generates a Python Egg (`.egg` file) for the project. Python Eggs are the preferred binary distribution format for EasyInstall, because they are cross-platform (for “pure” packages), directly importable, and contain project metadata including scripts and information about the project's dependencies. They can be simply downloaded and added to `sys.path` directly, or they can be placed in a directory on `sys.path` and then automatically discovered by the egg runtime system.

This command runs the `egg_info` command (if it hasn't already run) to update the project's metadata (`.egg-info`) directory. If you have added any extra metadata files to the `.egg-info` directory, those files will be included in the new egg file's metadata directory, for use by the egg runtime system or by any applications or frameworks that use that metadata.

You won't usually need to specify any special options for this command; just use `bdist_egg` and you're done. But there are a few options that may be occasionally useful:

**--dist-dir=DIR, -d DIR** Set the directory where the `.egg` file will be placed. If you don't supply this, then the `--dist-dir` setting of the `bdist` command will be used, which is usually a directory named `dist` in the project directory.

**--plat-name=PLATFORM, -p PLATFORM** Set the platform name string that will be embedded in the egg's filename (assuming the egg contains C extensions). This can be used to override the distutils default platform name with something more meaningful. Keep in mind, however, that the egg runtime system expects to see eggs with distutils platform names, so it may ignore or reject eggs with non-standard platform names. Similarly, the EasyInstall program may ignore them when searching web pages for download links. However, if you are cross-compiling or doing some other unusual things, you might find a use for this option.



**--exclude-source-files** Don't include any modules' `.py` files in the egg, just compiled Python, C, and data files. (Note that this doesn't affect any `.py` files in the EGG-INFO directory or its subdirectories, since for example there may be scripts with a `.py` extension which must still be retained.) We don't recommend that you use this option except for packages that are being bundled for proprietary end-user applications, or for "embedded" scenarios where space is at an absolute premium. On the other hand, if your package is going to be installed and used in compressed form, you might as well exclude the source because Python's `traceback` module doesn't currently understand how to display zipped source code anyway, or how to deal with files that are in a different place from where their code was compiled.

There are also some options you will probably never need, but which are there because they were copied from similar `bdist` commands used as an example for creating this one. They may be useful for testing and debugging, however, which is why we kept them:

**--keep-temp, -k** Keep the contents of the `--bdist-dir` tree around after creating the `.egg` file.

**--bdist-dir=DIR, -b DIR** Set the temporary directory for creating the distribution. The entire contents of this directory are zipped to create the `.egg` file, after running various installation commands to copy the package's modules, data, and extensions here.

**--skip-build** Skip doing any "build" commands; just go straight to the install-and-compress phases.

### 1.2.3 `develop` - Deploy the project source in "Development Mode"

This command allows you to deploy your project's source for use in one or more "staging areas" where it will be available for importing. This deployment is done in such a way that changes to the project source are immediately available in the staging area(s), without needing to run a build or install step after each change.

The `develop` command works by creating an `.egg-link` file (named for the project) in the given staging area. If the staging area is Python's `site-packages` directory, it also updates an `easy-install.pth` file so that the project is on `sys.path` by default for all programs run using that Python installation.

The `develop` command also installs wrapper scripts in the staging area (or a separate directory, as specified) that will ensure the project's dependencies are available on `sys.path` before running the project's source scripts. And, it ensures that any missing project dependencies are available in the staging area, by downloading and installing them if necessary.

Last, but not least, the `develop` command invokes the `build_ext -i` command to ensure any C extensions in the project have been built and are up-to-date, and the `egg_info` command to ensure the project's metadata is updated (so that the runtime and wrappers know what the project's dependencies are). If you make any changes to the project's setup script or C extensions, you should rerun the `develop` command against all relevant staging areas to keep the project's scripts, metadata and extensions up-to-date. Most other kinds of changes to your project should not require any build operations or rerunning `develop`, but keep in mind that even minor changes to the setup script (e.g. changing an entry point definition) require you to re-run the `develop` or `test` commands to keep the distribution updated.

Here are some of the options that the `develop` command accepts. Note that they affect the project's dependencies as well as the project itself, so if you have dependencies that need to be installed and you use `--exclude-scripts` (for example), the dependencies' scripts will not be installed either! For this reason, you may want to use `pip` to install the project's dependencies before using the `develop` command, if you need finer control over the installation options for dependencies.

**--uninstall, -u** Un-deploy the current project. You may use the `--install-dir` or `-d` option to designate the staging area. The created `.egg-link` file will be removed, if present and it is still pointing to the project directory. The project directory will be removed from `easy-install.pth` if the staging area is Python's `site-packages` directory.

Note that this option currently does *not* uninstall script wrappers! You must uninstall them yourself, or overwrite them by using `pip` to install a different version of the package. You can also avoid installing script wrappers in

the first place, if you use the `--exclude-scripts` (aka `-x`) option when you run `develop` to deploy the project.

**--multi-version, -m** “Multi-version” mode. Specifying this option prevents `develop` from adding an `easy-install.pth` entry for the project(s) being deployed, and if an entry for any version of a project already exists, the entry will be removed upon successful deployment. In multi-version mode, no specific version of the package is available for importing, unless you use `pkg_resources.require()` to put it on `sys.path`, or you are running a wrapper script generated by `setuptools`. (In which case the wrapper script calls `require()` for you.)

Note that if you install to a directory other than `site-packages`, this option is automatically in effect, because `.pth` files can only be used in `site-packages` (at least in Python 2.3 and 2.4). So, if you use the `--install-dir` or `-d` option (or they are set via configuration file(s)) your project and its dependencies will be deployed in multi-version mode.

**--install-dir=DIR, -d DIR** Set the installation directory (staging area). If this option is not directly specified on the command line or in a `distutils` configuration file, the `distutils` default installation location is used. Normally, this will be the `site-packages` directory, but if you are using `distutils` configuration files, setting things like `prefix` or `install_lib`, then those settings are taken into account when computing the default staging area.

**--script-dir=DIR, -s DIR** Set the script installation directory. If you don’t supply this option (via the command line or a configuration file), but you *have* supplied an `--install-dir` (via command line or config file), then this option defaults to the same directory, so that the scripts will be able to find their associated package installation. Otherwise, this setting defaults to the location where the `distutils` would normally install scripts, taking any `distutils` configuration file settings into account.

**--exclude-scripts, -x** Don’t deploy script wrappers. This is useful if you don’t want to disturb existing versions of the scripts in the staging area.

**--always-copy, -a** Copy all needed distributions to the staging area, even if they are already present in another directory on `sys.path`. By default, if a requirement can be met using a distribution that is already available in a directory on `sys.path`, it will not be copied to the staging area.

**--egg-path=DIR** Force the generated `.egg-link` file to use a specified relative path to the source directory. This can be useful in circumstances where your installation directory is being shared by code running under multiple platforms (e.g. Mac and Windows) which have different absolute locations for the code under development, but the same *relative* locations with respect to the installation directory. If you use this option when installing, you must supply the same relative path when uninstalling.

In addition to the above options, the `develop` command also accepts all of the same options accepted by `easy_install`. If you’ve configured any `easy_install` settings in your `setup.cfg` (or other `distutils` config files), the `develop` command will use them as defaults, unless you override them in a `[develop]` section or on the command line.

## 1.2.4 egg\_info - Create egg metadata and set build tags

This command performs two operations: it updates a project’s `.egg-info` metadata directory (used by the `bdist_egg`, `develop`, and `test` commands), and it allows you to temporarily change a project’s version string, to support “daily builds” or “snapshot” releases. It is run automatically by the `sdist`, `bdist_egg`, `develop`, and `test` commands in order to update the project’s metadata, but you can also specify it explicitly in order to temporarily change the project’s version string while executing other commands. (It also generates the `.egg-info/SOURCES.txt` manifest file, which is used when you are building source distributions.)

In addition to writing the core egg metadata defined by `setuptools` and required by `pkg_resources`, this command can be extended to write other metadata files as well, by defining entry points in the `egg_info.writers` group. See the section on [Adding new EGG-INFO Files](#) below for more details. Note that using additional metadata



writers may require you to include a `setup_requires` argument to `setup()` in order to ensure that the desired writers are available on `sys.path`.

## Release Tagging Options

The following options can be used to modify the project’s version string for all remaining commands on the `setup` command line. The options are processed in the order shown, so if you use more than one, the requested tags will be added in the following order:

**--tag-build=NAME, -b NAME** Append NAME to the project’s version string. Due to the way setuptools processes “pre-release” version suffixes beginning with the letters “a” through “e” (like “alpha”, “beta”, and “candidate”), you will usually want to use a tag like “build” or “.dev”, as this will cause the version number to be considered *lower* than the project’s default version. (If you want to make the version number *higher* than the default version, you can always leave off `--tag-build` and then use one or both of the following options.)

If you have a default build tag set in your `setup.cfg`, you can suppress it on the command line using `-b ""` or `--tag-build=""` as an argument to the `egg_info` command.

**--tag-date, -d** Add a date stamp of the form “-YYYYMMDD” (e.g. “-20050528”) to the project’s version number.

**--no-date, -D** Don’t include a date stamp in the version number. This option is included so you can override a default setting in `setup.cfg`.

(Note: Because these options modify the version number used for source and binary distributions of your project, you should first make sure that you know how the resulting version numbers will be interpreted by automated tools like `pip`. See the section above on *Specifying Your Project’s Version* for an explanation of pre- and post-release tags, as well as tips on how to choose and verify a versioning scheme for your project.)

For advanced uses, there is one other option that can be set, to change the location of the project’s `.egg-info` directory. Commands that need to find the project’s source directory or metadata should get it from this setting:

## Other egg\_info Options

**--egg-base=SOURCEDIR, -e SOURCEDIR** Specify the directory that should contain the `.egg-info` directory. This should normally be the root of your project’s source tree (which is not necessarily the same as your project directory; some projects use a `src` or `lib` subdirectory as the source root). You should not normally need to specify this directory, as it is normally determined from the `package_dir` argument to the `setup()` function, if any. If there is no `package_dir` set, this option defaults to the current directory.

## egg\_info Examples

Creating a dated “nightly build” snapshot egg:

```
setup.py egg_info --tag-date --tag-build=DEV bdist_egg
```

Creating a release with no version tags, even if some default tags are specified in `setup.cfg`:

```
setup.py egg_info -RDb "" sdist bdist_egg
```

(Notice that `egg_info` must always appear on the command line *before* any commands that you want the version changes to apply to.)

### 1.2.5 rotate - Delete outdated distribution files

As you develop new versions of your project, your distribution (`dist`) directory will gradually fill up with older source and/or binary distribution files. The `rotate` command lets you automatically clean these up, keeping only the `N` most-recently modified files matching a given pattern.

**--match=PATTERNLIST, -m PATTERNLIST** Comma-separated list of glob patterns to match. This option is *required*. The project name and `*` is prepended to the supplied patterns, in order to match only distributions belonging to the current project (in case you have a shared distribution directory for multiple projects). Typically, you will use a glob pattern like `.zip` or `.egg` to match files of the specified type. Note that each supplied pattern is treated as a distinct group of files for purposes of selecting files to delete.

**--keep=COUNT, -k COUNT** Number of matching distributions to keep. For each group of files identified by a pattern specified with the `--match` option, delete all but the `COUNT` most-recently-modified files in that group. This option is *required*.

**--dist-dir=DIR, -d DIR** Directory where the distributions are. This defaults to the value of the `bdist` command's `--dist-dir` option, which will usually be the project's `dist` subdirectory.

**Example 1:** Delete all `.tar.gz` files from the distribution directory, except for the 3 most recently modified ones:

```
setup.py rotate --match=.tar.gz --keep=3
```

**Example 2:** Delete all Python 2.3 or Python 2.4 eggs from the distribution directory, except the most recently modified one for each Python version:

```
setup.py rotate --match=py2.3*.egg,py2.4*.egg --keep=1
```

### 1.2.6 saveopts - Save used options to a configuration file

Finding and editing `distutils` configuration files can be a pain, especially since you also have to translate the configuration options from command-line form to the proper configuration file format. You can avoid these hassles by using the `saveopts` command. Just add it to the command line to save the options you used. For example, this command builds the project using the `mingw32` C compiler, then saves the `--compiler` setting as the default for future builds (even those run implicitly by the `install` command):

```
setup.py build --compiler=mingw32 saveopts
```

The `saveopts` command saves all options for every command specified on the command line to the project's local `setup.cfg` file, unless you use one of the [configuration file options](#) to change where the options are saved. For example, this command does the same as above, but saves the compiler setting to the site-wide (global) `distutils` configuration:

```
setup.py build --compiler=mingw32 saveopts -g
```

Note that it doesn't matter where you place the `saveopts` command on the command line; it will still save all the options specified for all commands. For example, this is another valid way to spell the last example:

```
setup.py saveopts -g build --compiler=mingw32
```

Note, however, that all of the commands specified are always run, regardless of where `saveopts` is placed on the command line.

## Configuration File Options

Normally, settings such as options and aliases are saved to the project's local `setup.cfg` file. But you can override this and save them to the global or per-user configuration files, or to a manually-specified filename.

**--global-config, -g** Save settings to the global `distutils.cfg` file inside the `distutils` package directory. You must have write access to that directory to use this option. You also can't combine this option with `-u` or `-f`.

**--user-config, -u** Save settings to the current user's `~/.pydistutils.cfg` (POSIX) or `$HOME/pydistutils.cfg` (Windows) file. You can't combine this option with `-g` or `-f`.

**--filename=FILENAME, -f FILENAME** Save settings to the specified configuration file to use. You can't combine this option with `-g` or `-u`. Note that if you specify a non-standard filename, the `distutils` and `setuptools` will not use the file's contents. This option is mainly included for use in testing.

These options are used by other `setuptools` commands that modify configuration files, such as the *alias* and *setopt* commands.

### 1.2.7 setopt - Set a distutils or setuptools option in a config file

This command is mainly for use by scripts, but it can also be used as a quick and dirty way to change a `distutils` configuration option without having to remember what file the options are in and then open an editor.

**Example 1.** Set the default C compiler to `mingw32` (using long option names):

```
setup.py setopt --command=build --option=compiler --set-value=mingw32
```

**Example 2.** Remove any setting for the `distutils` default package installation directory (short option names):

```
setup.py setopt -c install -o install_lib -r
```

Options for the `setopt` command:

**--command=COMMAND, -c COMMAND** Command to set the option for. This option is required.

**--option=OPTION, -o OPTION** The name of the option to set. This option is required.

**--set-value=VALUE, -s VALUE** The value to set the option to. Not needed if `-r` or `--remove` is set.

**--remove, -r** Remove (unset) the option, instead of setting it.

In addition to the above options, you may use any of the *configuration file options* (listed under the *saveopts* command, above) to determine which `distutils` configuration file the option will be added to (or removed from).

### 1.2.8 test - Build package and run a unittest suite

**Warning:** `test` is deprecated and will be removed in a future version. Users looking for a generic test entry point independent of test runner are encouraged to use `tox`.

When doing test-driven development, or running automated builds that need testing before they are deployed for downloading or use, it's often useful to be able to run a project's unit tests without actually deploying the project anywhere, even using the `develop` command. The `test` command runs a project's unit tests without actually deploying it, by temporarily putting the project's source on `sys.path`, after first running `build_ext -i` and `egg_info` to ensure that any C extensions and project metadata are up-to-date.

To use this command, your project's tests must be wrapped in a `unittest` test suite by either a function, a `TestCase` class or method, or a module or package containing `TestCase` classes. If the named suite is a module, and the module has an `additional_tests()` function, it is called and the result (which must be a `unittest.TestSuite`) is added to the tests to be run. If the named suite is a package, any submodules and subpackages are recursively added to the overall test suite. (Note: if your project specifies a `test_loader`, the rules for processing the chosen `test_suite` may differ; see the [test\\_loader](#) documentation for more details.)

Note that many test systems including `doctest` support wrapping their non-`unittest` tests in `TestSuite` objects. So, if you are using a test package that does not support this, we suggest you encourage its developers to implement test suite support, as this is a convenient and standard way to aggregate a collection of tests to be run under a common test harness.

By default, tests will be run in the “verbose” mode of the `unittest` package's text test runner, but you can get the “quiet” mode (just dots) if you supply the `-q` or `--quiet` option, either as a global option to the setup script (e.g. `setup.py -q test`) or as an option for the `test` command itself (e.g. `setup.py test -q`). There is one other option available:

**--test-suite=NAME, -s NAME** Specify the test suite (or module, class, or method) to be run (e.g. `some_module.test_suite`). The default for this option can be set by giving a `test_suite` argument to the `setup()` function, e.g.:

```
setup(
    # ...
    test_suite="my_package.tests.test_all"
)
```

If you did not set a `test_suite` in your `setup()` call, and do not provide a `--test-suite` option, an error will occur.

New in 41.5.0: Deprecated the `test` command.

## 1.2.9 upload - Upload source and/or egg distributions to PyPI

The `upload` command was deprecated in version 40.0 and removed in version 42.0. Use [twine](#) instead.

For more information on the current best practices in uploading your packages to PyPI, see the Python Packaging User Guide's “Packaging Python Projects” tutorial specifically the section on [uploading the distribution archives](#).

## 1.3 Configuring setup() using setup.cfg files

---

**Note:** New in 30.3.0 (8 Dec 2016).

---

---

**Important:** A `setup.py` file containing a `setup()` function call is still required even if your configuration resides in `setup.cfg`.

---

Setuptools allows using configuration files (usually `setup.cfg`) to define a package's metadata and other options that are normally supplied to the `setup()` function (declarative config).

This approach not only allows automation scenarios but also reduces boilerplate code in some cases.

---

**Note:** This implementation has limited compatibility with the `distutils2`-like `setup.cfg` sections used by the `pbr` and `d2to1` packages.

---

Namely: only metadata-related keys from metadata section are supported (except for description-file); keys from files, entry\_points and backwards\_compat are not supported.

```
[metadata]
name = my_package
version = attr: src.VERSION
description = My package description
long_description = file: README.rst, CHANGELOG.rst, LICENSE.rst
keywords = one, two
license = BSD 3-Clause License
classifiers =
    Framework :: Django
    License :: OSI Approved :: BSD License
    Programming Language :: Python :: 3
    Programming Language :: Python :: 3.5

[options]
zip_safe = False
include_package_data = True
packages = find:
scripts =
    bin/first.py
    bin/second.py
install_requires =
    requests
    importlib; python_version == "2.6"

[options.package_data]
* = *.txt, *.rst
hello = *.msg

[options.extras_require]
pdf = ReportLab>=1.2; RXP
rest = docutils>=0.3; pack ==1.1, ==1.3

[options.packages.find]
exclude =
    src.subpackage1
    src.subpackage2

[options.data_files]
/etc/my_package =
    site.d/00_default.conf
    host.d/00_default.conf
data = data/img/logo.png, data/svg/icon.svg
```

Metadata and options are set in the config sections of the same name.

- Keys are the same as the keyword arguments one provides to the `setup()` function.
- Complex values can be written comma-separated or placed one per line in *dangling* config values. The following are equivalent:

```
[metadata]
keywords = one, two

[metadata]
```

(continues on next page)

(continued from previous page)

```
keywords =  
    one  
    two
```

- In some cases, complex values can be provided in dedicated subsections for clarity.
- Some keys allow `file:`, `attr:`, and `find:` and `find_namespace:` directives in order to cover common usecases.
- Unknown keys are ignored.

### 1.3.1 Using a `src/` layout

One commonly used package configuration has all the module source code in a subdirectory (often called the `src/` layout), like this:

```
├── src  
│   └── mypackage  
│       ├── __init__.py  
│       └── mod1.py  
├── setup.py  
└── setup.cfg
```

You can set up your `setup.cfg` to automatically find all your packages in the subdirectory like this:

```
# This example contains just the necessary options for a src-layout, set up  
# the rest of the file as described above.  
  
[options]  
package_dir=  
    =src  
packages=find:  
  
[options.packages.find]  
where=src
```

### 1.3.2 Specifying values

Some values are treated as simple strings, some allow more logic.

Type names used below:

- `str` - simple string
- `list-comma` - dangling list or string of comma-separated values
- `list-semi` - dangling list or string of semicolon-separated values
- `bool` - True is 1, yes, true
- `dict` - list-comma where keys are separated from values by `=`
- `section` - values are read from a dedicated (sub)section

Special directives:

- `attr:` - Value is read from a module attribute. `attr:` supports callables and iterables; unsupported types are cast using `str()`.

- `file:` - Value is read from a list of files and then concatenated

---

**Note:** The `file:` directive is sandboxed and won't reach anything outside the directory containing `setup.py`.

---

## Metadata

---

**Note:** The aliases given below are supported for compatibility reasons, but their use is not advised.

---

Key	Aliases	Type	Minimum Version	Notes
<code>name</code>		str		
<code>version</code>		attr:, file:, str	39.2.0	(1)
<code>url</code>	<code>home-page</code>	str		
<code>download_url</code>	<code>download-url</code>	str		
<code>project_urls</code>		dict	38.3.0	
<code>author</code>		str		
<code>author_email</code>	<code>author-email</code>	str		
<code>maintainer</code>		str		
<code>maintainer_email</code>	<code>maintainer-email</code>	str		
<code>classifiers</code>	<code>classifier</code>	file:, list-comma		
<code>license</code>		str		
<code>license_file</code>		str		
<code>license_files</code>		list-comma		
<code>description</code>	<code>summary</code>	file:, str		
<code>long_description</code>	<code>long-description</code>	file:, str		
<code>long_description_content_type</code>		str	38.6.0	
<code>keywords</code>		list-comma		
<code>platforms</code>	<code>platform</code>	list-comma		
<code>provides</code>		list-comma		
<code>requires</code>		list-comma		
<code>obsoletes</code>		list-comma		

---

**Note:** A version loaded using the `file:` directive must comply with PEP 440. It is easy to accidentally put something other than a valid version string in such a file, so validation is stricter in this case.

---

Notes: 1. The `version` file attribute has only been supported since 39.2.0.

## Options

Key	Type	Minimum Version	Notes
zip_safe	bool		
setup_requires	list-semi		
install_requires	list-semi		
extras_require	section		
python_requires	str		
entry_points	file:, section		
use_2to3	bool		
use_2to3_fixers	list-comma		
use_2to3_exclude_fixers	list-comma		
convert_2to3_doctests	list-comma		
scripts	list-comma		
eager_resources	list-comma		
dependency_links	list-comma		
tests_require	list-semi		
include_package_data	bool		
packages	find:, find_namespace:, list-comma		
package_dir	dict		
package_data	section		(1)
exclude_package_data	section		
namespace_packages	list-comma		
py_modules	list-comma		
data_files	dict	40.6.0	

---

**Note:** `packages` - The `find:` and `find_namespace:` directive can be further configured in a dedicated subsection `options.packages.find`. This subsection accepts the same keys as the `setuptools.find_packages` and the `setuptools.find_namespace_packages` function: `where`, `include`, and `exclude`.

**find\_namespace directive** - The `find_namespace:` directive is supported since Python `>=3.3`.

---

Notes: 1. In the `package_data` section, a key named with a single asterisk (\*) refers to all packages, in lieu of the empty string used in `setup.py`.

### 1.3.3 Configuration API

Some automation tools may wish to access data from a configuration file.

Setuptools exposes a `read_configuration()` function for parsing metadata and options sections into a dictionary.

```
from setuptools.config import read_configuration

conf_dict = read_configuration("/home/user/dev/package/setup.cfg")
```

By default, `read_configuration()` will read only the file provided in the first argument. To include values from other configuration files which could be in various places, set the `find_others` keyword argument to `True`.



If you have only a configuration file but not the whole package, you can still try to get data out of it with the help of the `ignore_option_errors` keyword argument. When it is set to `True`, all options with errors possibly produced by directives, such as `attr:` and others, will be silently ignored. As a consequence, the resulting dictionary will include no such options.

## 1.4 Extending and Reusing Setuptools

### 1.4.1 Creating `distutils` Extensions

It can be hard to add new commands or setup arguments to the `distutils`. But the `setuptools` package makes it a bit easier, by allowing you to distribute a `distutils` extension as a separate project, and then have projects that need the extension just refer to it in their `setup_requires` argument.

With `setuptools`, your `distutils` extension projects can hook in new commands and `setup()` arguments just by defining “entry points”. These are mappings from command or argument names to a specification of where to import a handler from. (See the section on *Dynamic Discovery of Services and Plugins* above for some more background on entry points.)

#### Adding Commands

You can add new `setup` commands by defining entry points in the `distutils.commands` group. For example, if you wanted to add a `foo` command, you might add something like this to your `distutils` extension project’s setup script:

```
setup(
    # ...
    entry_points={
        "distutils.commands": [
            "foo = mypackage.some_module:foo",
        ],
    },
)
```

(Assuming, of course, that the `foo` class in `mypackage.some_module` is a `setuptools.Command` subclass.)

Once a project containing such entry points has been activated on `sys.path`, (e.g. by running “install” or “develop” with a site-packages installation directory) the command(s) will be available to any `setuptools`-based setup scripts. It is not necessary to use the `--command-packages` option or to monkeypatch the `distutils.command` package to install your commands; `setuptools` automatically adds a wrapper to the `distutils` to search for entry points in the active distributions on `sys.path`. In fact, this is how `setuptools`’ own commands are installed: the `setuptools` project’s setup script defines entry points for them!

#### Adding `setup()` Arguments

**Warning:** Adding arguments to `setup` is discouraged as such arguments are only supported through imperative execution and not supported through declarative config.

Sometimes, your commands may need additional arguments to the `setup()` call. You can enable this by defining entry points in the `distutils.setup_keywords` group. For example, if you wanted a `setup()` argument called `bar_baz`, you might add something like this to your `distutils` extension project’s setup script:

```
setup(
    # ...
    entry_points={
        "distutils.commands": [
            "foo = mypackage.some_module:foo",
        ],
        "distutils.setup_keywords": [
            "bar_baz = mypackage.some_module:validate_bar_baz",
        ],
    },
)
```

The idea here is that the entry point defines a function that will be called to validate the `setup()` argument, if it's supplied. The `Distribution` object will have the initial value of the attribute set to `None`, and the validation function will only be called if the `setup()` call sets it to a non-`None` value. Here's an example validation function:

```
def assert_bool(dist, attr, value):
    """Verify that value is True, False, 0, or 1"""
    if bool(value) != value:
        raise DistutilsSetupError(
            "%r must be a boolean value (got %r)" % (attr, value)
        )
```

Your function should accept three arguments: the `Distribution` object, the attribute name, and the attribute value. It should raise a `DistutilsSetupError` (from the `distutils.errors` module) if the argument is invalid. Remember, your function will only be called with non-`None` values, and the default value of arguments defined this way is always `None`. So, your commands should always be prepared for the possibility that the attribute will be `None` when they access it later.

If more than one active distribution defines an entry point for the same `setup()` argument, *all* of them will be called. This allows multiple `distutils` extensions to define a common argument, as long as they agree on what values of that argument are valid.

Also note that as with commands, it is not necessary to subclass or monkeypatch the `distutils Distribution` class in order to add your arguments; it is sufficient to define the entry points in your extension, as long as any setup script using your extension lists your project in its `setup_requires` argument.

## Customizing Distribution Options

Plugins may wish to extend or alter the options on a `Distribution` object to suit the purposes of that project. For example, a tool that infers the `Distribution.version` from SCM-metadata may need to hook into the option finalization. To enable this feature, Setuptools offers an entry point “`setuptools.finalize_distribution_options`”. That entry point must be a callable taking one argument (the `Distribution` instance).

If the callable has an `.order` property, that value will be used to determine the order in which the hook is called. Lower numbers are called first and the default is zero (0).

Plugins may read, alter, and set properties on the distribution, but each plugin is encouraged to load the configuration/settings for their behavior independently.

## Adding new EGG-INFO Files

Some extensible applications or frameworks may want to allow third parties to develop plugins with application or framework-specific metadata included in the plugins' EGG-INFO directory, for easy access via the `pkg_resources` metadata API. The easiest way to allow this is to create a `distutils` extension to be used from the plugin projects' setup

scripts (via `setup_requires`) that defines a new setup keyword, and then uses that data to write an EGG-INFO file when the `egg_info` command is run.

The `egg_info` command looks for extension points in an `egg_info.writers` group, and calls them to write the files. Here's a simple example of a `distutils` extension defining a setup argument `foo_bar`, which is a list of lines that will be written to `foo_bar.txt` in the EGG-INFO directory of any project that uses the argument:

```
setup(
    # ...
    entry_points={
        "distutils.setup_keywords": [
            "foo_bar = setuptools.dist:assert_string_list",
        ],
        "egg_info.writers": [
            "foo_bar.txt = setuptools.command.egg_info:write_arg",
        ],
    },
)
```

This simple example makes use of two utility functions defined by `setuptools` for its own use: a routine to validate that a setup keyword is a sequence of strings, and another one that looks up a setup argument and writes it to a file. Here's what the writer utility looks like:

```
def write_arg(cmd, basename, filename):
    argname = os.path.splitext(basename)[0]
    value = getattr(cmd.distribution, argname, None)
    if value is not None:
        value = "\n".join(value) + "\n"
    cmd.write_or_delete_file(argname, filename, value)
```

As you can see, `egg_info.writers` entry points must be a function taking three arguments: a `egg_info` command instance, the basename of the file to write (e.g. `foo_bar.txt`), and the actual full filename that should be written to.

In general, writer functions should honor the command object's `dry_run` setting when writing files, and use the `distutils.log` object to do any console output. The easiest way to conform to this requirement is to use the `cmd` object's `write_file()`, `delete_file()`, and `write_or_delete_file()` methods exclusively for your file operations. See those methods' docstrings for more details.

## Adding Support for Revision Control Systems

If the files you want to include in the source distribution are tracked using Git, Mercurial or SVN, you can use the following packages to achieve that:

- Git and Mercurial: `setuptools_scm`
- SVN: `setuptools_svn`

If you would like to create a plugin for `setuptools` to find files tracked by another revision control system, you can do so by adding an entry point to the `setuptools.file_finders` group. The entry point should be a function accepting a single directory name, and should yield all the filenames within that directory (and any subdirectories thereof) that are under revision control.

For example, if you were going to create a plugin for a revision control system called “foobar”, you would write a function something like this:

```
def find_files_for_foobar(dirname):
    # loop to yield paths that start with `dirname`
```

And you would register it in a setup script using something like this:

```
entry_points={
    "setuptools.file_finders": [
        "foobar = my_foobar_module:find_files_for_foobar",
    ]
}
```

Then, anyone who wants to use your plugin can simply install it, and their local setuptools installation will be able to find the necessary files.

It is not necessary to distribute source control plugins with projects that simply use the other source control system, or to specify the plugins in `setup_requires`. When you create a source distribution with the `sdist` command, setuptools automatically records what files were found in the `SOURCES.txt` file. That way, recipients of source distributions don't need to have revision control at all. However, if someone is working on a package by checking out with that system, they will need the same plugin(s) that the original author is using.

A few important points for writing revision control file finders:

- Your finder function **MUST** return relative paths, created by appending to the passed-in directory name. Absolute paths are **NOT** allowed, nor are relative paths that reference a parent directory of the passed-in directory.
- Your finder function **MUST** accept an empty string as the directory name, meaning the current directory. You **MUST NOT** convert this to a dot; just yield relative paths. So, yielding a subdirectory named `some/dir` under the current directory should **NOT** be rendered as `./some/dir` or `/somewhere/some/dir`, but *always* as simply `some/dir`
- Your finder function **SHOULD NOT** raise any errors, and **SHOULD** deal gracefully with the absence of needed programs (i.e., ones belonging to the revision control system itself. It *may*, however, use `distutils.log.warn()` to inform the user of the missing program(s).

### 1.4.2 Mailing List and Bug Tracker

Please use the [distutils-sig mailing list](#) for questions and discussion about setuptools, and the [setuptools bug tracker](#) **ONLY** for issues you have confirmed via the list are actual bugs, and which you have reduced to a minimal set of steps to reproduce.

---

# Package Discovery and Resource Access using `pkg_resources`

---

The `pkg_resources` module distributed with `setuptools` provides an API for Python libraries to access their resource files, and for extensible applications and frameworks to automatically discover plugins. It also provides runtime support for using C extensions that are inside zipfile-format eggs, support for merging packages that have separately-distributed modules or subpackages, and APIs for managing Python’s current “working set” of active packages.

### Table of Contents

- *Package Discovery and Resource Access using `pkg_resources`*
  - *Overview*
  - *API Reference*
    - \* *Namespace Package Support*
    - \* *WorkingSet Objects*
      - *Basic WorkingSet Methods*
      - *WorkingSet Methods and Attributes*
      - *Receiving Change Notifications*
      - *Locating Plugins*
    - \* *Environment Objects*
    - \* *Requirement Objects*
      - *Requirements Parsing*
      - *Requirement Methods and Attributes*
    - \* *Entry Points*
      - *Convenience API*

- *Creating and Parsing*
- *EntryPoint Objects*
- \* *Distribution Objects*
  - *Getting or Creating Distributions*
  - *Distribution Attributes*
  - *Distribution Methods*
- \* *ResourceManager API*
  - *Basic Resource Access*
  - *Resource Extraction*
  - *“Provider” Interface*
- \* *Metadata API*
  - *IMetadataProvider Methods*
- \* *Exceptions*
- \* *Supporting Custom Importers*
  - *IResourceProvider*
  - *Built-in Resource Providers*
- \* *Utility Functions*
  - *Parsing Utilities*
  - *Platform Utilities*
  - *PEP 302 Utilities*
  - *File/Path Utilities*
  - *History*

## 2.1 Overview

The `pkg_resources` module provides runtime facilities for finding, introspecting, activating and using installed Python distributions. Some of the more advanced features (notably the support for parallel installation of multiple versions) rely specifically on the “egg” format (either as a zip archive or subdirectory), while others (such as plugin discovery) will work correctly so long as “egg-info” metadata directories are available for relevant distributions.

Eggs are a distribution format for Python modules, similar in concept to Java’s “jars” or Ruby’s “gems”, or the “wheel” format defined in PEP 427. However, unlike a pure distribution format, eggs can also be installed and added directly to `sys.path` as an import location. When installed in this way, eggs are *discoverable*, meaning that they carry metadata that unambiguously identifies their contents and dependencies. This means that an installed egg can be *automatically* found and added to `sys.path` in response to simple requests of the form, “get me everything I need to use docutils’ PDF support”. This feature allows mutually conflicting versions of a distribution to co-exist in the same Python installation, with individual applications activating the desired version at runtime by manipulating the contents of `sys.path` (this differs from the virtual environment approach, which involves creating isolated environments for each application).

The following terms are needed in order to explain the capabilities offered by this module:

**project** A library, framework, script, plugin, application, or collection of data or other resources, or some combination thereof. Projects are assumed to have “relatively unique” names, e.g. names registered with PyPI.

**release** A snapshot of a project at a particular point in time, denoted by a version identifier.

**distribution** A file or files that represent a particular release.

**importable distribution** A file or directory that, if placed on `sys.path`, allows Python to import any modules contained within it.

**pluggable distribution** An importable distribution whose filename unambiguously identifies its release (i.e. project and version), and whose contents unambiguously specify what releases of other projects will satisfy its runtime requirements.

**extra** An “extra” is an optional feature of a release, that may impose additional runtime requirements. For example, if docutils PDF support required a PDF support library to be present, docutils could define its PDF support as an “extra”, and list what other project releases need to be available in order to provide it.

**environment** A collection of distributions potentially available for importing, but not necessarily active. More than one distribution (i.e. release version) for a given project may be present in an environment.

**working set** A collection of distributions actually available for importing, as on `sys.path`. At most one distribution (release version) of a given project may be present in a working set, as otherwise there would be ambiguity as to what to import.

**eggs** Eggs are pluggable distributions in one of the three formats currently supported by `pkg_resources`. There are built eggs, development eggs, and egg links. Built eggs are directories or zipfiles whose name ends with `.egg` and follows the egg naming conventions, and contain an `EGG-INFO` subdirectory (zipped or otherwise). Development eggs are normal directories of Python code with one or more `ProjectName.egg-info` subdirectories. The development egg format is also used to provide a default version of a distribution that is available to software that doesn’t use `pkg_resources` to request specific versions. Egg links are `*.egg-link` files that contain the name of a built or development egg, to support symbolic linking on platforms that do not have native symbolic links (or where the symbolic link support is limited).

(For more information about these terms and concepts, see also this [architectural overview](#) of `pkg_resources` and Python Eggs in general.)

## 2.2 API Reference

### 2.2.1 Namespace Package Support

A namespace package is a package that only contains other packages and modules, with no direct contents of its own. Such packages can be split across multiple, separately-packaged distributions. They are normally used to split up large packages produced by a single organization, such as in the `zope` namespace package for Zope Corporation packages, and the `peak` namespace package for the Python Enterprise Application Kit.

To create a namespace package, you list it in the `namespace_packages` argument to `setup()`, in your project’s `setup.py`. (See the [setuptools documentation on namespace packages](#) for more information on this.) Also, you must add a `declare_namespace()` call in the package’s `__init__.py` file(s):

**declare\_namespace(name)** Declare that the dotted package name *name* is a “namespace package” whose contained packages and modules may be spread across multiple distributions. The named package’s `__path__` will be extended to include the corresponding package in all distributions on `sys.path` that contain a package of that name. (More precisely, if an importer’s `find_module(name)` returns a loader, then it will also be searched for the package’s contents.) Whenever a Distribution’s `activate()` method is invoked, it checks for the presence of namespace packages and updates their `__path__` contents accordingly.

Applications that manipulate namespace packages or directly alter `sys.path` at runtime may also need to use this API function:

**fixup\_namespace\_packages(*path\_item*)** Declare that *path\_item* is a newly added item on `sys.path` that may need to be used to update existing namespace packages. Ordinarily, this is called for you when an egg is automatically added to `sys.path`, but if your application modifies `sys.path` to include locations that may contain portions of a namespace package, you will need to call this function to ensure they are added to the existing namespace packages.

Although by default `pkg_resources` only supports namespace packages for filesystem and zip importers, you can extend its support to other “importers” compatible with PEP 302 using the `register_namespace_handler()` function. See the section below on [Supporting Custom Importers](#) for details.

## 2.2.2 WorkingSet Objects

The `WorkingSet` class provides access to a collection of “active” distributions. In general, there is only one meaningful `WorkingSet` instance: the one that represents the distributions that are currently active on `sys.path`. This global instance is available under the name `working_set` in the `pkg_resources` module. However, specialized tools may wish to manipulate working sets that don’t correspond to `sys.path`, and therefore may wish to create other `WorkingSet` instances.

It’s important to note that the global `working_set` object is initialized from `sys.path` when `pkg_resources` is first imported, but is only updated if you do all future `sys.path` manipulation via `pkg_resources` APIs. If you manually modify `sys.path`, you must invoke the appropriate methods on the `working_set` instance to keep it in sync. Unfortunately, Python does not provide any way to detect arbitrary changes to a list object like `sys.path`, so `pkg_resources` cannot automatically update the `working_set` based on changes to `sys.path`.

**WorkingSet(*entries=None*)** Create a `WorkingSet` from an iterable of path entries. If *entries* is not supplied, it defaults to the value of `sys.path` at the time the constructor is called.

Note that you will not normally construct `WorkingSet` instances yourself, but instead you will implicitly or explicitly use the global `working_set` instance. For the most part, the `pkg_resources` API is designed so that the `working_set` is used by default, such that you don’t have to explicitly refer to it most of the time.

All distributions available directly on `sys.path` will be activated automatically when `pkg_resources` is imported. This behaviour can cause version conflicts for applications which require non-default versions of those distributions. To handle this situation, `pkg_resources` checks for a `__requires__` attribute in the `__main__` module when initializing the default working set, and uses this to ensure a suitable version of each affected distribution is activated. For example:

```
__requires__ = ["CherryPy < 3"] # Must be set before pkg_resources import
import pkg_resources
```

### Basic WorkingSet Methods

The following methods of `WorkingSet` objects are also available as module-level functions in `pkg_resources` that apply to the default `working_set` instance. Thus, you can use e.g. `pkg_resources.require()` as an abbreviation for `pkg_resources.working_set.require()`:

**require(\**requirements*)** Ensure that distributions matching *requirements* are activated

*requirements* must be a string or a (possibly-nested) sequence thereof, specifying the distributions and versions required. The return value is a sequence of the distributions that needed to be activated to fulfill the requirements; all relevant distributions are included, even if they were already activated in this working set.

For the syntax of requirement specifiers, see the section below on [Requirements Parsing](#).



In general, it should not be necessary for you to call this method directly. It's intended more for use in quick-and-dirty scripting and interactive interpreter hacking than for production use. If you're creating an actual library or application, it's strongly recommended that you create a "setup.py" script using `setuptools`, and declare all your requirements there. That way, tools like `pip` can automatically detect what requirements your package has, and deal with them accordingly.

Note that calling `require('SomePackage')` will not install `SomePackage` if it isn't already present. If you need to do this, you should use the `resolve()` method instead, which allows you to pass an `installer` callback that will be invoked when a needed distribution can't be found on the local machine. You can then have this callback display a dialog, automatically download the needed distribution, or whatever else is appropriate for your application. See the documentation below on the `resolve()` method for more information, and also on the `obtain()` method of `Environment` objects.

**`run_script(requires, script_name)`** Locate distribution specified by *requires* and run its *script\_name* script. *requires* must be a string containing a requirement specifier. (See [Requirements Parsing](#) below for the syntax.)

The script, if found, will be executed in *the caller's globals*. That's because this method is intended to be called from wrapper scripts that act as a proxy for the "real" scripts in a distribution. A wrapper script usually doesn't need to do anything but invoke this function with the correct arguments.

If you need more control over the script execution environment, you probably want to use the `run_script()` method of a `Distribution` object's [Metadata API](#) instead.

**`iter_entry_points(group, name=None)`** Yield entry point objects from *group* matching *name*

If *name* is `None`, yields all entry points in *group* from all distributions in the working set, otherwise only ones matching both *group* and *name* are yielded. Entry points are yielded from the active distributions in the order that the distributions appear in the working set. (For the global `working_set`, this should be the same as the order that they are listed in `sys.path`.) Note that within the entry points advertised by an individual distribution, there is no particular ordering.

Please see the section below on [Entry Points](#) for more information.

## WorkingSet Methods and Attributes

These methods are used to query or manipulate the contents of a specific working set, so they must be explicitly invoked on a particular `WorkingSet` instance:

**`add_entry(entry)`** Add a path item to the `entries`, finding any distributions on it. You should use this when you add additional items to `sys.path` and you want the global `working_set` to reflect the change. This method is also called by the `WorkingSet()` constructor during initialization.

This method uses `find_distributions(entry, True)` to find distributions corresponding to the path entry, and then `add()` them. *entry* is always appended to the `entries` attribute, even if it is already present, however. (This is because `sys.path` can contain the same value more than once, and the `entries` attribute should be able to reflect this.)

**`__contains__(dist)`** True if *dist* is active in this `WorkingSet`. Note that only one distribution for a given project can be active in a given `WorkingSet`.

**`__iter__()`** Yield distributions for non-duplicate projects in the working set. The yield order is the order in which the items' path entries were added to the working set.

**`find(req)`** Find a distribution matching *req* (a `Requirement` instance). If there is an active distribution for the requested project, this returns it, as long as it meets the version requirement specified by *req*. But, if there is an active distribution for the project and it does *not* meet the *req* requirement, `VersionConflict` is raised. If there is no active distribution for the requested project, `None` is returned.

**resolve(requirements, env=None, installer=None)** List all distributions needed to (recursively) meet *requirements*

*requirements* must be a sequence of `Requirement` objects. *env*, if supplied, should be an `Environment` instance. If not supplied, an `Environment` is created from the working set's `entries`. *installer*, if supplied, will be invoked with each requirement that cannot be met by an already-installed distribution; it should return a `Distribution` or `None`. (See the `obtain()` method of *Environment Objects*, below, for more information on the *installer* argument.)

**add(dist, entry=None)** Add *dist* to working set, associated with *entry*

If *entry* is unspecified, it defaults to `dist.location`. On exit from this routine, *entry* is added to the end of the working set's `.entries` (if it wasn't already present).

*dist* is only added to the working set if it's for a project that doesn't already have a distribution active in the set. If it's successfully added, any callbacks registered with the `subscribe()` method will be called. (See *Receiving Change Notifications*, below.)

Note: `add()` is automatically called for you by the `require()` method, so you don't normally need to use this method directly.

**entries** This attribute represents a “shadow” `sys.path`, primarily useful for debugging. If you are experiencing import problems, you should check the global `working_set` object's `entries` against `sys.path`, to ensure that they match. If they do not, then some part of your program is manipulating `sys.path` without updating the `working_set` accordingly. IMPORTANT NOTE: do not directly manipulate this attribute! Setting it equal to `sys.path` will not fix your problem, any more than putting black tape over an “engine warning” light will fix your car! If this attribute is out of sync with `sys.path`, it's merely an *indicator* of the problem, not the cause of it.

## Receiving Change Notifications

Extensible applications and frameworks may need to receive notification when a new distribution (such as a plug-in component) has been added to a working set. This is what the `subscribe()` method and `add_activation_listener()` function are for.

**subscribe(callback)** Invoke `callback(distribution)` once for each active distribution that is in the set now, or gets added later. Because the callback is invoked for already-active distributions, you do not need to loop over the working set yourself to deal with the existing items; just register the callback and be prepared for the fact that it will be called immediately by this method.

Note that callbacks *must not* allow exceptions to propagate, or they will interfere with the operation of other callbacks and possibly result in an inconsistent working set state. Callbacks should use a `try/except` block to ignore, log, or otherwise process any errors, especially since the code that caused the callback to be invoked is unlikely to be able to handle the errors any better than the callback itself.

`pkg_resources.add_activation_listener()` is an alternate spelling of `pkg_resources.working_set.subscribe()`.

## Locating Plugins

Extensible applications will sometimes have a “plugin directory” or a set of plugin directories, from which they want to load entry points or other metadata. The `find_plugins()` method allows you to do this, by scanning an environment for the newest version of each project that can be safely loaded without conflicts or missing requirements.

**find\_plugins(plugin\_env, full\_env=None, fallback=True)** Scan *plugin\_env* and identify which distributions could be added to this working set without version conflicts or missing requirements.

Example usage:

```
distributions, errors = working_set.find_plugins(
    Environment(plugin_dirlist)
)
map(working_set.add, distributions) # add plugins+libs to sys.path
print "Couldn't load", errors      # display errors
```

The *plugin\_env* should be an `Environment` instance that contains only distributions that are in the project's "plugin directory" or directories. The *full\_env*, if supplied, should be an `Environment` instance that contains all currently-available distributions.

If *full\_env* is not supplied, one is created automatically from the `WorkingSet` this method is called on, which will typically mean that every directory on `sys.path` will be scanned for distributions.

This method returns a 2-tuple: (*distributions*, *error\_info*), where *distributions* is a list of the distributions found in *plugin\_env* that were loadable, along with any other distributions that are needed to resolve their dependencies. *error\_info* is a dictionary mapping unloadable plugin distributions to an exception instance describing the error that occurred. Usually this will be a `DistributionNotFound` or `VersionConflict` instance.

Most applications will use this method mainly on the master `working_set` instance in `pkg_resources`, and then immediately add the returned distributions to the working set so that they are available on `sys.path`. This will make it possible to find any entry points, and allow any other metadata tracking and hooks to be activated.

The resolution algorithm used by `find_plugins()` is as follows. First, the project names of the distributions present in *plugin\_env* are sorted. Then, each project's eggs are tried in descending version order (i.e., newest version first).

An attempt is made to resolve each egg's dependencies. If the attempt is successful, the egg and its dependencies are added to the output list and to a temporary copy of the working set. The resolution process continues with the next project name, and no older eggs for that project are tried.

If the resolution attempt fails, however, the error is added to the error dictionary. If the *fallback* flag is true, the next older version of the plugin is tried, until a working version is found. If false, the resolution process continues with the next plugin project name.

Some applications may have stricter fallback requirements than others. For example, an application that has a database schema or persistent objects may not be able to safely downgrade a version of a package. Others may want to ensure that a new plugin configuration is either 100% good or else revert to a known-good configuration. (That is, they may wish to revert to a known configuration if the *error\_info* return value is non-empty.)

Note that this algorithm gives precedence to satisfying the dependencies of alphabetically prior project names in case of version conflicts. If two projects named "AaronsPlugin" and "ZekesPlugin" both need different versions of "TomsLibrary", then "AaronsPlugin" will win and "ZekesPlugin" will be disabled due to version conflict.

### 2.2.3 Environment Objects

An "environment" is a collection of `Distribution` objects, usually ones that are present and potentially importable on the current platform. `Environment` objects are used by `pkg_resources` to index available distributions during dependency resolution.

**Environment(search\_path=None, platform=get\_supported\_platform(), python=PY\_MAJOR)**

Create an environment snapshot by scanning *search\_path* for distributions compatible with *platform* and *python*. *search\_path* should be a sequence of strings such as might be used on `sys.path`. If a *search\_path* isn't supplied, `sys.path` is used.

*platform* is an optional string specifying the name of the platform that platform-specific distributions must be compatible with. If unspecified, it defaults to the current platform. *python* is an optional string naming the desired version of Python (e.g. '2.4'); it defaults to the currently-running version.

You may explicitly set *platform* (and/or *python*) to `None` if you wish to include *all* distributions, not just those compatible with the running platform or Python version.

Note that *search\_path* is scanned immediately for distributions, and the resulting `Environment` is a snapshot of the found distributions. It is not automatically updated if the system's state changes due to e.g. installation or removal of distributions.

**`__getitem__(project_name)`** Returns a list of distributions for the given project name, ordered from newest to oldest version. (And highest to lowest format precedence for distributions that contain the same version of the project.) If there are no distributions for the project, returns an empty list.

**`__iter__()`** Yield the unique project names of the distributions in this environment. The yielded names are always in lower case.

**`add(dist)`** Add *dist* to the environment if it matches the platform and python version specified at creation time, and only if the distribution hasn't already been added. (i.e., adding the same distribution more than once is a no-op.)

**`remove(dist)`** Remove *dist* from the environment.

**`can_add(dist)`** Is distribution *dist* acceptable for this environment? If it's not compatible with the *platform* and *python* version values specified when the environment was created, a false value is returned.

**`__add__(dist_or_env) (+ operator)`** Add a distribution or environment to an `Environment` instance, returning a *new* environment object that contains all the distributions previously contained by both. The new environment will have a *platform* and *python* of `None`, meaning that it will not reject any distributions from being added to it; it will simply accept whatever is added. If you want the added items to be filtered for platform and Python version, or you want to add them to the *same* environment instance, you should use in-place addition (`+=`) instead.

**`__iadd__(dist_or_env) (+= operator)`** Add a distribution or environment to an `Environment` instance *in-place*, updating the existing instance and returning it. The *platform* and *python* filter attributes take effect, so distributions in the source that do not have a suitable platform string or Python version are silently ignored.

**`best_match(req, working_set, installer=None)`** Find distribution best matching *req* and usable on *working\_set*

This calls the `find(req)` method of the *working\_set* to see if a suitable distribution is already active. (This may raise `VersionConflict` if an unsuitable version of the project is already active in the specified *working\_set*.) If a suitable distribution isn't active, this method returns the newest distribution in the environment that meets the Requirement in *req*. If no suitable distribution is found, and *installer* is supplied, then the result of calling the environment's `obtain(req, installer)` method will be returned.

**`obtain(requirement, installer=None)`** Obtain a distro that matches requirement (e.g. via download). In the base `Environment` class, this routine just returns `installer(requirement)`, unless *installer* is `None`, in which case `None` is returned instead. This method is a hook that allows subclasses to attempt other ways of obtaining a distribution before falling back to the *installer* argument.

**`scan(search_path=None)`** Scan *search\_path* for distributions usable on *platform*

Any distributions found are added to the environment. *search\_path* should be a sequence of strings such as might be used on `sys.path`. If not supplied, `sys.path` is used. Only distributions conforming to the platform/python version defined at initialization are added. This method is a shortcut for using the `find_distributions()` function to find the distributions from each item in *search\_path*, and then calling `add()` to add each one to the environment.

## 2.2.4 Requirement Objects

Requirement objects express what versions of a project are suitable for some purpose. These objects (or their string form) are used by various `pkg_resources` APIs in order to find distributions that a script or distribution needs.

## Requirements Parsing

**parse\_requirements(s)** Yield `Requirement` objects for a string or iterable of lines. Each requirement must start on a new line. See below for syntax.

**Requirement.parse(s)** Create a `Requirement` object from a string or iterable of lines. A `ValueError` is raised if the string or lines do not contain a valid requirement specifier, or if they contain more than one specifier. (To parse multiple specifiers from a string or iterable of strings, use `parse_requirements()` instead.)

The syntax of a requirement specifier is defined in full in PEP 508.

Some examples of valid requirement specifiers:

```
FooProject >= 1.2
Fizzy [foo, bar]
PickyThing<1.6,>1.9,!1.9.6,<2.0a0,==2.4c1
SomethingWhoseVersionIDontCareAbout
SomethingWithMarker[foo]>1.0;python_version<"2.7"
```

The project name is the only required portion of a requirement string, and if it's the only thing supplied, the requirement will accept any version of that project.

The “extras” in a requirement are used to request optional features of a project, that may require additional project distributions in order to function. For example, if the hypothetical “Report-O-Rama” project offered optional PDF support, it might require an additional library in order to provide that support. Thus, a project needing Report-O-Rama’s PDF features could use a requirement of `Report-O-Rama[PDF]` to request installation or activation of both Report-O-Rama and any libraries it needs in order to provide PDF support. For example, you could use:

```
pip install Report-O-Rama[PDF]
```

To install the necessary packages using `pip`, or call `pkg_resources.require('Report-O-Rama[PDF]')` to add the necessary distributions to `sys.path` at runtime.

The “markers” in a requirement are used to specify when a requirement should be installed – the requirement will be installed if the marker evaluates as true in the current environment. For example, specifying `argparse;python_version<"3.0"` will not install in an Python 3 environment, but will in a Python 2 environment.

## Requirement Methods and Attributes

**\_\_contains\_\_(dist\_or\_version)** Return true if *dist\_or\_version* fits the criteria for this requirement. If *dist\_or\_version* is a `Distribution` object, its project name must match the requirement’s project name, and its version must meet the requirement’s version criteria. If *dist\_or\_version* is a string, it is parsed using the `parse_version()` utility function. Otherwise, it is assumed to be an already-parsed version.

The `Requirement` object’s version specifiers (`.specs`) are internally sorted into ascending version order, and used to establish what ranges of versions are acceptable. Adjacent redundant conditions are effectively consolidated (e.g. `">1, >2"` produces the same results as `">2"`, and `"<2, <3"` produces the same results as `"<2"`). `"!="` versions are excised from the ranges they fall within. The version being tested for acceptability is then checked for membership in the resulting ranges.

**\_\_eq\_\_(other\_requirement)** A requirement compares equal to another requirement if they have case-insensitively equal project names, version specifiers, and “extras”. (The order that extras and version specifiers are in is also ignored.) Equal requirements also have equal hashes, so that requirements can be used in sets or as dictionary keys.

**\_\_str\_\_()** The string form of a `Requirement` is a string that, if passed to `Requirement.parse()`, would return an equal `Requirement` object.

**project\_name** The name of the required project

**key** An all-lowercase version of the `project_name`, useful for comparison or indexing.

**extras** A tuple of names of “extras” that this requirement calls for. (These will be all-lowercase and normalized using the `safe_extra()` parsing utility function, so they may not exactly equal the extras the requirement was created with.)

**specs** A list of `(op, version)` tuples, sorted in ascending parsed-version order. The *op* in each tuple is a comparison operator, represented as a string. The *version* is the (unparsed) version number.

**marker** An instance of `packaging.markers.Marker` that allows evaluation against the current environment. May be `None` if no marker specified.

**url** The location to download the requirement from if specified.

## 2.2.5 Entry Points

Entry points are a simple way for distributions to “advertise” Python objects (such as functions or classes) for use by other distributions. Extensible applications and frameworks can search for entry points with a particular name or group, either from a specific distribution or from all active distributions on `sys.path`, and then inspect or load the advertised objects at will.

Entry points belong to “groups” which are named with a dotted name similar to a Python package or module name. For example, the `setuptools` package uses an entry point named `distutils.commands` in order to find commands defined by `distutils` extensions. `setuptools` treats the names of entry points defined in that group as the acceptable commands for a setup script.

In a similar way, other packages can define their own entry point groups, either using dynamic names within the group (like `distutils.commands`), or possibly using predefined names within the group. For example, a blogging framework that offers various pre- or post-publishing hooks might define an entry point group and look for entry points named “pre\_process” and “post\_process” within that group.

To advertise an entry point, a project needs to use `setuptools` and provide an `entry_points` argument to `setup()` in its setup script, so that the entry points will be included in the distribution’s metadata. For more details, see the [setuptools documentation](<https://setuptools.readthedocs.io/en/latest/setuptools.html#dynamic-discovery-of-services-and-plugins>).

Each project distribution can advertise at most one entry point of a given name within the same entry point group. For example, a `distutils` extension could advertise two different `distutils.commands` entry points, as long as they had different names. However, there is nothing that prevents *different* projects from advertising entry points of the same name in the same group. In some cases, this is a desirable thing, since the application or framework that uses the entry points may be calling them as hooks, or in some other way combining them. It is up to the application or framework to decide what to do if multiple distributions advertise an entry point; some possibilities include using both entry points, displaying an error message, using the first one found in `sys.path` order, etc.

## Convenience API

In the following functions, the *dist* argument can be a `Distribution` instance, a `Requirement` instance, or a string specifying a requirement (i.e. project name, version, etc.). If the argument is a string or `Requirement`, the specified distribution is located (and added to `sys.path` if not already present). An error will be raised if a matching distribution is not available.

The *group* argument should be a string containing a dotted identifier, identifying an entry point group. If you are defining an entry point group, you should include some portion of your package’s name in the group name so as to avoid collision with other packages’ entry point groups.



**load\_entry\_point(dist, group, name)** Load the named entry point from the specified distribution, or raise `ImportError`.

**get\_entry\_info(dist, group, name)** Return an `EntryPoint` object for the given *group* and *name* from the specified distribution. Returns `None` if the distribution has not advertised a matching entry point.

**get\_entry\_map(dist, group=None)** Return the distribution's entry point map for *group*, or the full entry map for the distribution. This function always returns a dictionary, even if the distribution advertises no entry points. If *group* is given, the dictionary maps entry point names to the corresponding `EntryPoint` object. If *group* is `None`, the dictionary maps group names to dictionaries that then map entry point names to the corresponding `EntryPoint` instance in that group.

**iter\_entry\_points(group, name=None)** Yield entry point objects from *group* matching *name*.

If *name* is `None`, yields all entry points in *group* from all distributions in the working set on `sys.path`, otherwise only ones matching both *group* and *name* are yielded. Entry points are yielded from the active distributions in the order that the distributions appear on `sys.path`. (Within entry points for a particular distribution, however, there is no particular ordering.)

(This API is actually a method of the global `working_set` object; see the section above on [Basic WorkingSet Methods](#) for more information.)

## Creating and Parsing

**EntryPoint(name, module\_name, attrs=(), extras=(), dist=None)** Create an `EntryPoint` instance. *name* is the entry point name. The *module\_name* is the (dotted) name of the module containing the advertised object. *attrs* is an optional tuple of names to look up from the module to obtain the advertised object. For example, an *attrs* of `("foo", "bar")` and a *module\_name* of `"baz"` would mean that the advertised object could be obtained by the following code:

```
import baz
advertised_object = baz.foo.bar
```

The *extras* are an optional tuple of “extra feature” names that the distribution needs in order to provide this entry point. When the entry point is loaded, these extra features are looked up in the *dist* argument to find out what other distributions may need to be activated on `sys.path`; see the `load()` method for more details. The *extras* argument is only meaningful if *dist* is specified. *dist* must be a `Distribution` instance.

**EntryPoint.parse(src, dist=None) (classmethod)** Parse a single entry point from string *src*

Entry point syntax follows the form:

```
name = some.module:some.attr [extral,extra2]
```

The entry name and module name are required, but the `:attrs` and `[extras]` parts are optional, as is the whitespace shown between some of the items. The *dist* argument is passed through to the `EntryPoint()` constructor, along with the other values parsed from *src*.

**EntryPoint.parse\_group(group, lines, dist=None) (classmethod)** Parse *lines* (a string or sequence of lines) to create a dictionary mapping entry point names to `EntryPoint` objects. `ValueError` is raised if entry point names are duplicated, if *group* is not a valid entry point group name, or if there are any syntax errors. (Note: the *group* parameter is used only for validation and to create more informative error messages.) If *dist* is provided, it will be used to set the *dist* attribute of the created `EntryPoint` objects.

**EntryPoint.parse\_map(data, dist=None) (classmethod)** Parse *data* into a dictionary mapping group names to dictionaries mapping entry point names to `EntryPoint` objects. If *data* is a dictionary, then the keys are used as group names and the values are passed to `parse_group()` as the *lines* argument. If *data* is a string or sequence of lines, it is first split into `.ini`-style sections (using the `split_sections()` utility

function) and the section names are used as group names. In either case, the *dist* argument is passed through to `parse_group()` so that the entry points will be linked to the specified distribution.

### EntryPoint Objects

For simple introspection, `EntryPoint` objects have attributes that correspond exactly to the constructor argument names: `name`, `module_name`, `attrs`, `extras`, and `dist` are all available. In addition, the following methods are provided:

**load()** Load the entry point, returning the advertised Python object. Effectively calls `self.require()` then returns `self.resolve()`.

**require(env=None, installer=None)** Ensure that any “extras” needed by the entry point are available on `sys.path`. `UnknownExtra` is raised if the `EntryPoint` has `extras`, but no `dist`, or if the named extras are not defined by the distribution. If *env* is supplied, it must be an `Environment`, and it will be used to search for needed distributions if they are not already present on `sys.path`. If *installer* is supplied, it must be a callable taking a `Requirement` instance and returning a matching importable `Distribution` instance or `None`.

**resolve()** Resolve the entry point from its module and `attrs`, returning the advertised Python object. Raises `ImportError` if it cannot be obtained.

**\_\_str\_\_()** The string form of an `EntryPoint` is a string that could be passed to `EntryPoint.parse()` to produce an equivalent `EntryPoint`.

## 2.2.6 Distribution Objects

`Distribution` objects represent collections of Python code that may or may not be importable, and may or may not have metadata and resources associated with them. Their metadata may include information such as what other projects the distribution depends on, what entry points the distribution advertises, and so on.

### Getting or Creating Distributions

Most commonly, you’ll obtain `Distribution` objects from a `WorkingSet` or an `Environment`. (See the sections above on *WorkingSet Objects* and *Environment Objects*, which are containers for active distributions and available distributions, respectively.) You can also obtain `Distribution` objects from one of these high-level APIs:

**find\_distributions(path\_item, only=False)** Yield distributions accessible via *path\_item*. If *only* is true, yield only distributions whose `location` is equal to *path\_item*. In other words, if *only* is true, this yields any distributions that would be importable if *path\_item* were on `sys.path`. If *only* is false, this also yields distributions that are “in” or “under” *path\_item*, but would not be importable unless their locations were also added to `sys.path`.

**get\_distribution(dist\_spec)** Return a `Distribution` object for a given `Requirement` or string. If *dist\_spec* is already a `Distribution` instance, it is returned. If it is a `Requirement` object or a string that can be parsed into one, it is used to locate and activate a matching distribution, which is then returned.

However, if you’re creating specialized tools for working with distributions, or creating a new distribution format, you may also need to create `Distribution` objects directly, using one of the three constructors below.

These constructors all take an optional *metadata* argument, which is used to access any resources or metadata associated with the distribution. *metadata* must be an object that implements the `IResourceProvider` interface, or `None`. If it is `None`, an `EmptyProvider` is used instead. `Distribution` objects implement both the *IResourceProvider* and *IMetadataProvider Methods* by delegating them to the *metadata* object.



**Distribution.from\_location(location, basename, metadata=None, \*\*kw)** (classmethod)

Create a distribution for *location*, which must be a string such as a URL, filename, or other string that might be used on `sys.path`. *basename* is a string naming the distribution, like `Foo-1.2-py2.4.egg`. If *basename* ends with `.egg`, then the project's name, version, python version and platform are extracted from the filename and used to set those properties of the created distribution. Any additional keyword arguments are forwarded to the `Distribution()` constructor.

**Distribution.from\_filename(filename, metadata=None\*\*kw)** (classmethod) Create a distribution by parsing a local filename. This is a shorter way of saying `Distribution.from_location(normalize_path(filename), os.path.basename(filename), metadata)`. In other words, it creates a distribution whose location is the normalize form of the filename, parsing name and version information from the base portion of the filename. Any additional keyword arguments are forwarded to the `Distribution()` constructor.

**Distribution(location, metadata, project\_name, version, py\_version, platform, precedence)**

Create a distribution by setting its properties. All arguments are optional and default to `None`, except for *py\_version* (which defaults to the current Python version) and *precedence* (which defaults to `EGG_DIST`; for more details see *precedence* under [Distribution Attributes](#) below). Note that it's usually easier to use the `from_filename()` or `from_location()` constructors than to specify all these arguments individually.

## Distribution Attributes

**location** A string indicating the distribution's location. For an importable distribution, this is the string that would be added to `sys.path` to make it actively importable. For non-importable distributions, this is simply a filename, URL, or other way of locating the distribution.

**project\_name** A string, naming the project that this distribution is for. Project names are defined by a project's setup script, and they are used to identify projects on PyPI. When a `Distribution` is constructed, the *project\_name* argument is passed through the `safe_name()` utility function to filter out any unacceptable characters.

**key** `dist.key` is short for `dist.project_name.lower()`. It's used for case-insensitive comparison and indexing of distributions by project name.

**extras** A list of strings, giving the names of extra features defined by the project's dependency list (the `extras_require` argument specified in the project's setup script).

**version** A string denoting what release of the project this distribution contains. When a `Distribution` is constructed, the *version* argument is passed through the `safe_version()` utility function to filter out any unacceptable characters. If no *version* is specified at construction time, then attempting to access this attribute later will cause the `Distribution` to try to discover its version by reading its `PKG-INFO` metadata file. If `PKG-INFO` is unavailable or can't be parsed, `ValueError` is raised.

**parsed\_version** The `parsed_version` is an object representing a "parsed" form of the distribution's version. `dist.parsed_version` is a shortcut for calling `parse_version(dist.version)`. It is used to compare or sort distributions by version. (See the [Parsing Utilities](#) section below for more information on the `parse_version()` function.) Note that accessing `parsed_version` may result in a `ValueError` if the `Distribution` was constructed without a *version* and without *metadata* capable of supplying the missing version info.

**py\_version** The major/minor Python version the distribution supports, as a string. For example, "2.7" or "3.4". The default is the current version of Python.

**platform** A string representing the platform the distribution is intended for, or `None` if the distribution is "pure Python" and therefore cross-platform. See [Platform Utilities](#) below for more information on platform strings.

**precedence** A distribution's *precedence* is used to determine the relative order of two distributions that have the same *project\_name* and *parsed\_version*. The default precedence is `pkg_resources.EGG_DIST`, which is the highest (i.e. most preferred) precedence. The full list of predefined precedences, from

most preferred to least preferred, is: `EGG_DIST`, `BINARY_DIST`, `SOURCE_DIST`, `CHECKOUT_DIST`, and `DEVELOP_DIST`. Normally, precedences other than `EGG_DIST` are used only by the `setuptools.package_index` module, when sorting distributions found in a package index to determine their suitability for installation. “System” and “Development” eggs (i.e., ones that use the `.egg-info` format), however, are automatically given a precedence of `DEVELOP_DIST`.

## Distribution Methods

**activate(path=None)** Ensure distribution is importable on *path*. If *path* is `None`, `sys.path` is used instead. This ensures that the distribution’s `location` is in the *path* list, and it also performs any necessary namespace package fixups or declarations. (That is, if the distribution contains namespace packages, this method ensures that they are declared, and that the distribution’s contents for those namespace packages are merged with the contents provided by any other active distributions. See the section above on [Namespace Package Support](#) for more information.)

`pkg_resources` adds a notification callback to the global `working_set` that ensures this method is called whenever a distribution is added to it. Therefore, you should not normally need to explicitly call this method. (Note that this means that namespace packages on `sys.path` are always imported as soon as `pkg_resources` is, which is another reason why namespace packages should not contain any code or import statements.)

**as\_requirement()** Return a `Requirement` instance that matches this distribution’s project name and version.

**requires(extras=())** List the `Requirement` objects that specify this distribution’s dependencies. If *extras* is specified, it should be a sequence of names of “extras” defined by the distribution, and the list returned will then include any dependencies needed to support the named “extras”.

**clone(\*\*kw)** Create a copy of the distribution. Any supplied keyword arguments override the corresponding argument to the `Distribution()` constructor, allowing you to change some of the copied distribution’s attributes.

**egg\_name()** Return what this distribution’s standard filename should be, not including the “.egg” extension. For example, a distribution for project “Foo” version 1.2 that runs on Python 2.3 for Windows would have an `egg_name()` of `Foo-1.2-py2.3-win32`. Any dashes in the name or version are converted to underscores. (`Distribution.from_location()` will convert them back when parsing a “.egg” file name.)

**\_\_cmp\_\_(other), \_\_hash\_\_()** Distribution objects are hashed and compared on the basis of their parsed version and precedence, followed by their key (lowercase project name), location, Python version, and platform.

The following methods are used to access `EntryPoint` objects advertised by the distribution. See the section above on [Entry Points](#) for more detailed information about these operations:

**get\_entry\_info(group, name)** Return the `EntryPoint` object for *group* and *name*, or `None` if no such point is advertised by this distribution.

**get\_entry\_map(group=None)** Return the entry point map for *group*. If *group* is `None`, return a dictionary mapping group names to entry point maps for all groups. (An entry point map is a dictionary of entry point names to `EntryPoint` objects.)

**load\_entry\_point(group, name)** Short for `get_entry_info(group, name).load()`. Returns the object advertised by the named entry point, or raises `ImportError` if the entry point isn’t advertised by this distribution, or there is some other import problem.

In addition to the above methods, `Distribution` objects also implement all of the [IResourceProvider](#) and [IMetadataProvider Methods](#) (which are documented in later sections):

- `has_metadata(name)`
- `metadata_isdir(name)`

- `metadata_listdir(name)`
- `get_metadata(name)`
- `get_metadata_lines(name)`
- `run_script(script_name, namespace)`
- `get_resource_filename(manager, resource_name)`
- `get_resource_stream(manager, resource_name)`
- `get_resource_string(manager, resource_name)`
- `has_resource(resource_name)`
- `resource_isdir(resource_name)`
- `resource_listdir(resource_name)`

If the distribution was created with a *metadata* argument, these resource and metadata access methods are all delegated to that *metadata* provider. Otherwise, they are delegated to an `EmptyProvider`, so that the distribution will appear to have no resources or metadata. This delegation approach is used so that supporting custom importers or new distribution formats can be done simply by creating an appropriate *IResourceProvider* implementation; see the section below on *Supporting Custom Importers* for more details.

## 2.2.7 ResourceManager API

The `ResourceManager` class provides uniform access to package resources, whether those resources exist as files and directories or are compressed in an archive of some kind.

Normally, you do not need to create or explicitly manage `ResourceManager` instances, as the `pkg_resources` module creates a global instance for you, and makes most of its methods available as top-level names in the `pkg_resources` module namespace. So, for example, this code actually calls the `resource_string()` method of the global `ResourceManager`:

```
import pkg_resources
my_data = pkg_resources.resource_string(__name__, "foo.dat")
```

Thus, you can use the APIs below without needing an explicit `ResourceManager` instance; just import and use them as needed.

### Basic Resource Access

In the following methods, the *package\_or\_requirement* argument may be either a Python package/module name (e.g. `foo.bar`) or a `Requirement` instance. If it is a package or module name, the named module or package must be importable (i.e., be in a distribution or directory on `sys.path`), and the *resource\_name* argument is interpreted relative to the named package. (Note that if a module name is used, then the resource name is relative to the package immediately containing the named module. Also, you should not use use a namespace package name, because a namespace package can be spread across multiple distributions, and is therefore ambiguous as to which distribution should be searched for the resource.)

If it is a `Requirement`, then the requirement is automatically resolved (searching the current `Environment` if necessary) and a matching distribution is added to the `WorkingSet` and `sys.path` if one was not already present. (Unless the `Requirement` can't be satisfied, in which case an exception is raised.) The *resource\_name* argument is then interpreted relative to the root of the identified distribution; i.e. its first path segment will be treated as a peer of the top-level modules or packages in the distribution.

Note that resource names must be `/`-separated paths rooted at the package, cannot contain relative names like `".."`, and cannot be absolute. Do *not* use `os.path` routines to manipulate resource paths, as they are *not* filesystem paths.

**`resource_exists(package_or_requirement, resource_name)`** Does the named resource exist? Return `True` or `False` accordingly.

**`resource_stream(package_or_requirement, resource_name)`** Return a readable file-like object for the specified resource; it may be an actual file, a `StringIO`, or some similar object. The stream is in “binary mode”, in the sense that whatever bytes are in the resource will be read as-is.

**`resource_string(package_or_requirement, resource_name)`** Return the specified resource as a string. The resource is read in binary fashion, such that the returned string contains exactly the bytes that are stored in the resource.

**`resource_isdir(package_or_requirement, resource_name)`** Is the named resource a directory? Return `True` or `False` accordingly.

**`resource_listdir(package_or_requirement, resource_name)`** List the contents of the named resource directory, just like `os.listdir` except that it works even if the resource is in a zipfile.

Note that only `resource_exists()` and `resource_isdir()` are insensitive as to the resource type. You cannot use `resource_listdir()` on a file resource, and you can’t use `resource_string()` or `resource_stream()` on directory resources. Using an inappropriate method for the resource type may result in an exception or undefined behavior, depending on the platform and distribution format involved.

## Resource Extraction

**`resource_filename(package_or_requirement, resource_name)`** Sometimes, it is not sufficient to access a resource in string or stream form, and a true filesystem filename is needed. In such cases, you can use this method (or module-level function) to obtain a filename for a resource. If the resource is in an archive distribution (such as a zipped egg), it will be extracted to a cache directory, and the filename within the cache will be returned. If the named resource is a directory, then all resources within that directory (including subdirectories) are also extracted. If the named resource is a C extension or “eager resource” (see the `setuptools` documentation for details), then all C extensions and eager resources are extracted at the same time.

Archived resources are extracted to a cache location that can be managed by the following two methods:

**`set_extraction_path(path)`** Set the base path where resources will be extracted to, if needed.

If you do not call this routine before any extractions take place, the path defaults to the return value of `get_default_cache()`. (Which is based on the `PYTHON_EGG_CACHE` environment variable, with various platform-specific fallbacks. See that routine’s documentation for more details.)

Resources are extracted to subdirectories of this path based upon information given by the resource provider. You may set this to a temporary directory, but then you must call `cleanup_resources()` to delete the extracted files when done. There is no guarantee that `cleanup_resources()` will be able to remove all extracted files. (On Windows, for example, you can’t unlink `.pyd` or `.dll` files that are still in use.)

Note that you may not change the extraction path for a given resource manager once resources have been extracted, unless you first call `cleanup_resources()`.

**`cleanup_resources(force=False)`** Delete all extracted resource files and directories, returning a list of the file and directory names that could not be successfully removed. This function does not have any concurrency protection, so it should generally only be called when the extraction path is a temporary directory exclusive to a single process. This method is not automatically called; you must call it explicitly or register it as an `atexit` function if you wish to ensure cleanup of a temporary directory used for extractions.

## “Provider” Interface

If you are implementing an `IResourceProvider` and/or `IMetadataProvider` for a new distribution archive format, you may need to use the following `IResourceManager` methods to co-ordinate extraction of resources to the filesystem. If you’re not implementing an archive format, however, you have no need to use these methods. Unlike the other methods listed above, they are *not* available as top-level functions tied to the global `ResourceManager`; you must therefore have an explicit `ResourceManager` instance to use them.

**get\_cache\_path(*archive\_name*, *names*=())** Return absolute location in cache for *archive\_name* and *names*

The parent directory of the resulting path will be created if it does not already exist. *archive\_name* should be the base filename of the enclosing egg (which may not be the name of the enclosing zipfile!), including its “.egg” extension. *names*, if provided, should be a sequence of path name parts “under” the egg’s extraction location.

This method should only be called by resource providers that need to obtain an extraction location, and only for names they intend to extract, as it tracks the generated names for possible cleanup later.

**extraction\_error()** Raise an `ExtractionError` describing the active exception as interfering with the extraction process. You should call this if you encounter any OS errors extracting the file to the cache path; it will format the operating system exception for you, and add other information to the `ExtractionError` instance that may be needed by programs that want to wrap or handle extraction errors themselves.

**postprocess(*tempname*, *filename*)** Perform any platform-specific postprocessing of *tempname*. Resource providers should call this method **ONLY** after successfully extracting a compressed resource. They must **NOT** call it on resources that are already in the filesystem.

*tempname* is the current (temporary) name of the file, and *filename* is the name it will be renamed to by the caller after this routine returns.

## 2.2.8 Metadata API

The metadata API is used to access metadata resources bundled in a pluggable distribution. Metadata resources are virtual files or directories containing information about the distribution, such as might be used by an extensible application or framework to connect “plugins”. Like other kinds of resources, metadata resource names are /-separated and should not contain `.` or begin with a `/`. You should not use `os.path` routines to manipulate resource paths.

The metadata API is provided by objects implementing the `IMetadataProvider` or `IResourceProvider` interfaces. Distribution objects implement this interface, as do objects returned by the `get_provider()` function:

**get\_provider(*package\_or\_requirement*)** If a package name is supplied, return an `IResourceProvider` for the package. If a `Requirement` is supplied, resolve it by returning a `Distribution` from the current working set (searching the current `Environment` if necessary and adding the newly found `Distribution` to the working set). If the named package can’t be imported, or the `Requirement` can’t be satisfied, an exception is raised.

NOTE: if you use a package name rather than a `Requirement`, the object you get back may not be a pluggable distribution, depending on the method by which the package was installed. In particular, “development” packages and “single-version externally-managed” packages do not have any way to map from a package name to the corresponding project’s metadata. Do not write code that passes a package name to `get_provider()` and then tries to retrieve project metadata from the returned object. It may appear to work when the named package is in an `.egg` file or directory, but it will fail in other installation scenarios. If you want project metadata, you need to ask for a *project*, not a package.

### IMetadataProvider Methods

The methods provided by objects (such as `Distribution` instances) that implement the `IMetadataProvider` or `IResourceProvider` interfaces are:

**has\_metadata(name)** Does the named metadata resource exist?

**metadata\_isdir(name)** Is the named metadata resource a directory?

**metadata\_listdir(name)** List of metadata names in the directory (like `os.listdir()`)

**get\_metadata(name)** Return the named metadata resource as a string. The data is read in binary mode; i.e., the exact bytes of the resource file are returned.

**get\_metadata\_lines(name)** Yield named metadata resource as list of non-blank non-comment lines. This is short for calling `yield_lines(provider.get_metadata(name))`. See the section on [yield\\_lines\(\)](#) below for more information on the syntax it recognizes.

**run\_script(script\_name, namespace)** Execute the named script in the supplied namespace dictionary. Raises `ResolutionError` if there is no script by that name in the `scripts` metadata directory. *namespace* should be a Python dictionary, usually a module dictionary if the script is being run as a module.

## 2.2.9 Exceptions

`pkg_resources` provides a simple exception hierarchy for problems that may occur when processing requests to locate and activate packages:

```
ResolutionError
  DistributionNotFound
  VersionConflict
  UnknownExtra

ExtractionError
```

**ResolutionError** This class is used as a base class for the other three exceptions, so that you can catch all of them with a single “except” clause. It is also raised directly for miscellaneous requirement-resolution problems like trying to run a script that doesn’t exist in the distribution it was requested from.

**DistributionNotFound** A distribution needed to fulfill a requirement could not be found.

**VersionConflict** The requested version of a project conflicts with an already-activated version of the same project.

**UnknownExtra** One of the “extras” requested was not recognized by the distribution it was requested from.

**ExtractionError** A problem occurred extracting a resource to the Python Egg cache. The following attributes are available on instances of this exception:

**manager** The resource manager that raised this exception

**cache\_path** The base directory for resource extraction

**original\_error** The exception instance that caused extraction to fail

### 2.2.10 Supporting Custom Importers

By default, `pkg_resources` supports normal filesystem imports, and `zipimport` importers. If you wish to use the `pkg_resources` features with other (PEP 302-compatible) importers or module loaders, you may need to register various handlers and support functions using these APIs:



**register\_finder(importer\_type, distribution\_finder)** Register *distribution\_finder* to find distributions in `sys.path` items. *importer\_type* is the type or class of a PEP 302 “Importer” (`sys.path` item handler), and *distribution\_finder* is a callable that, when passed a path item, the importer instance, and an *only* flag, yields `Distribution` instances found under that path item. (The *only* flag, if true, means the finder should yield only `Distribution` objects whose `location` is equal to the path item provided.)

See the source of the `pkg_resources.find_on_path` function for an example finder function.

**register\_loader\_type(loader\_type, provider\_factory)** Register *provider\_factory* to make `IResourceProvider` objects for *loader\_type*. *loader\_type* is the type or class of a PEP 302 module. `__loader__`, and *provider\_factory* is a function that, when passed a module object, returns an *IResourceProvider* for that module, allowing it to be used with the *ResourceManager API*.

**register\_namespace\_handler(importer\_type, namespace\_handler)** Register *namespace\_handler* to declare namespace packages for the given *importer\_type*. *importer\_type* is the type or class of a PEP 302 “importer” (`sys.path` item handler), and *namespace\_handler* is a callable with a signature like this:

```
def namespace_handler(importer, path_entry, moduleName, module):
    # return a path_entry to use for child packages
```

Namespace handlers are only called if the relevant importer object has already agreed that it can handle the relevant path item. The handler should only return a subpath if the module `__path__` does not already contain an equivalent subpath. Otherwise, it should return `None`.

For an example namespace handler, see the source of the `pkg_resources.file_ns_handler` function, which is used for both zipfile importing and regular importing.

## IResourceProvider

`IResourceProvider` is an abstract class that documents what methods are required of objects returned by a *provider\_factory* registered with `register_loader_type()`. `IResourceProvider` is a subclass of `IMetadataProvider`, so objects that implement this interface must also implement all of the *IMetadataProvider Methods* as well as the methods shown here. The *manager* argument to the methods below must be an object that supports the full *ResourceManager API* documented above.

**get\_resource\_filename(manager, resource\_name)** Return a true filesystem path for *resource\_name*, coordinating the extraction with *manager*, if the resource must be unpacked to the filesystem.

**get\_resource\_stream(manager, resource\_name)** Return a readable file-like object for *resource\_name*.

**get\_resource\_string(manager, resource\_name)** Return a string containing the contents of *resource\_name*.

**has\_resource(resource\_name)** Does the package contain the named resource?

**resource\_isdir(resource\_name)** Is the named resource a directory? Return a false value if the resource does not exist or is not a directory.

**resource\_listdir(resource\_name)** Return a list of the contents of the resource directory, ala `os.listdir()`. Requesting the contents of a non-existent directory may raise an exception.

Note, by the way, that your provider classes need not (and should not) subclass `IResourceProvider` or `IMetadataProvider`! These classes exist solely for documentation purposes and do not provide any useful implementation code. You may instead wish to subclass one of the *built-in resource providers*.

## Built-in Resource Providers

`pkg_resources` includes several provider classes that are automatically used where appropriate. Their inheritance tree looks like this:

```
NullProvider
  EggProvider
    DefaultProvider
      PathMetadata
      ZipProvider
        EggMetadata
    EmptyProvider
      FileMetadata
```

**NullProvider** This provider class is just an abstract base that provides for common provider behaviors (such as running scripts), given a definition for just a few abstract methods.

**EggProvider** This provider class adds in some egg-specific features that are common to zipped and unzipped eggs.

**DefaultProvider** This provider class is used for unpacked eggs and “plain old Python” filesystem modules.

**ZipProvider** This provider class is used for all zipped modules, whether they are eggs or not.

**EmptyProvider** This provider class always returns answers consistent with a provider that has no metadata or resources. Distribution objects created without a `metadata` argument use an instance of this provider class instead. Since all `EmptyProvider` instances are equivalent, there is no need to have more than one instance. `pkg_resources` therefore creates a global instance of this class under the name `empty_provider`, and you may use it if you have need of an `EmptyProvider` instance.

**PathMetadata(*path*, *egg\_info*)** Create an `IResourceProvider` for a filesystem-based distribution, where *path* is the filesystem location of the importable modules, and *egg\_info* is the filesystem location of the distribution’s metadata directory. *egg\_info* should usually be the EGG-INFO subdirectory of *path* for an “unpacked egg”, and a `ProjectName.egg-info` subdirectory of *path* for a “development egg”. However, other uses are possible for custom purposes.

**EggMetadata(*zipimporter*)** Create an `IResourceProvider` for a zipfile-based distribution. The *zipimporter* should be a `zipimport.zipimporter` instance, and may represent a “basket” (a zipfile containing multiple “.egg” subdirectories) a specific egg *within* a basket, or a zipfile egg (where the zipfile itself is a “.egg”). It can also be a combination, such as a zipfile egg that also contains other eggs.

**FileMetadata(*path\_to\_pkg\_info*)** Create an `IResourceProvider` that provides exactly one metadata resource: PKG-INFO. The supplied path should be a distutils PKG-INFO file. This is basically the same as an `EmptyProvider`, except that requests for PKG-INFO will be answered using the contents of the designated file. (This provider is used to wrap `.egg-info` files installed by vendor-supplied system packages.)

## 2.2.11 Utility Functions

In addition to its high-level APIs, `pkg_resources` also includes several generally-useful utility routines. These routines are used to implement the high-level APIs, but can also be quite useful by themselves.

### Parsing Utilities

**parse\_version(*version*)** Parsed a project’s version string as defined by PEP 440. The returned value will be an object that represents the version. These objects may be compared to each other and sorted. The sorting algorithm is as defined by PEP 440 with the addition that any version which is not a valid PEP 440 version will be considered less than any valid PEP 440 version and the invalid versions will continue sorting using the original algorithm.



**yield\_lines(strs)** Yield non-empty/non-comment lines from a string/unicode or a possibly- nested sequence thereof. If *strs* is an instance of `basestring`, it is split into lines, and each non-blank, non-comment line is yielded after stripping leading and trailing whitespace. (Lines whose first non-blank character is `#` are considered comment lines.)

If *strs* is not an instance of `basestring`, it is iterated over, and each item is passed recursively to `yield_lines()`, so that an arbitrarily nested sequence of strings, or sequences of sequences of strings can be flattened out to the lines contained therein. So for example, passing a file object or a list of strings to `yield_lines` will both work. (Note that between each string in a sequence of strings there is assumed to be an implicit line break, so lines cannot bridge two strings in a sequence.)

This routine is used extensively by `pkg_resources` to parse metadata and file formats of various kinds, and most other `pkg_resources` parsing functions that yield multiple values will use it to break up their input. However, this routine is idempotent, so calling `yield_lines()` on the output of another call to `yield_lines()` is completely harmless.

**split\_sections(strs)** Split a string (or possibly-nested iterable thereof), yielding (`section`, `content`) pairs found using an `.ini`-like syntax. Each `section` is a whitespace-stripped version of the section name ("`[section]`") and each `content` is a list of stripped lines excluding blank lines and comment-only lines. If there are any non-blank, non-comment lines before the first section header, they're yielded in a first `section` of `None`.

This routine uses `yield_lines()` as its front end, so you can pass in anything that `yield_lines()` accepts, such as an open text file, string, or sequence of strings. `ValueError` is raised if a malformed section header is found (i.e. a line starting with `[` but not ending with `]`).

Note that this simplistic parser assumes that any line whose first nonblank character is `[` is a section heading, so it can't support `.ini` format variations that allow `[` as the first nonblank character on other lines.

**safe\_name(name)** Return a "safe" form of a project's name, suitable for use in a Requirement string, as a distribution name, or a PyPI project name. All non-alphanumeric runs are condensed to single `-` characters, such that a name like "The \$\$\$ Tree" becomes "The-Tree". Note that if you are generating a filename from this value you should combine it with a call to `to_filename()` so all dashes ("`-`") are replaced by underscores ("`_`"). See `to_filename()`.

**safe\_version(version)** This will return the normalized form of any PEP 440 version, if the version string is not PEP 440 compatible than it is similar to `safe_name()` except that spaces in the input become dots, and dots are allowed to exist in the output. As with `safe_name()`, if you are generating a filename from this you should replace any `-` characters in the output with underscores.

**safe\_extra(extra)** Return a "safe" form of an extra's name, suitable for use in a requirement string or a setup script's `extras_require` keyword. This routine is similar to `safe_name()` except that non-alphanumeric runs are replaced by a single underbar (`_`), and the result is lowercased.

**to\_filename(name\_or\_version)** Escape a name or version string so it can be used in a dash-separated filename (or `#egg=name-version` tag) without ambiguity. You should only pass in values that were returned by `safe_name()` or `safe_version()`.

## Platform Utilities

**get\_build\_platform()** Return this platform's identifier string. For Windows, the return value is `"win32"`, and for Mac OS X it is a string of the form `"macosx-10.4-ppc"`. All other platforms return the same uname-based string that the `distutils.util.get_platform()` function returns. This string is the minimum platform version required by distributions built on the local machine. (Backward compatibility note: setuptools versions prior to 0.6b1 called this function `get_platform()`, and the function is still available under that name for backward compatibility reasons.)

**get\_supported\_platform()** (New in 0.6b1) This is similar to `get_build_platform()`, but is the maximum platform version that the local machine supports. You will usually want to use this value as the provided argument to the `compatible_platforms()` function.

**compatible\_platforms(provided, required)** Return true if a distribution built on the *provided* platform may be used on the *required* platform. If either platform value is `None`, it is considered a wildcard, and the platforms are therefore compatible. Likewise, if the platform strings are equal, they're also considered compatible, and `True` is returned. Currently, the only non-equal platform strings that are considered compatible are Mac OS X platform strings with the same hardware type (e.g. `ppc`) and major version (e.g. `10`) with the *provided* platform's minor version being less than or equal to the *required* platform's minor version.

**get\_default\_cache()** Determine the default cache location for extracting resources from zipped eggs. This routine returns the `PYTHON_EGG_CACHE` environment variable, if set. Otherwise, on Windows, it returns a "Python-Eggs" subdirectory of the user's "Application Data" directory. On all other systems, it returns `os.path.expanduser("~/python-eggs")` if `PYTHON_EGG_CACHE` is not set.

## PEP 302 Utilities

**get\_importer(path\_item)** A deprecated alias for `pkgutil.get_importer()`

## File/Path Utilities

**ensure\_directory(path)** Ensure that the parent directory (`os.path.dirname()` of *path*) actually exists, using `os.makedirs()` if necessary.

**normalize\_path(path)** Return a "normalized" version of *path*, such that two paths represent the same filesystem location if they have equal `normalized_path()` values. Specifically, this is a shortcut for calling `os.path.realpath` and `os.path.normcase` on *path*. Unfortunately, on certain platforms (notably Cygwin and Mac OS X) the `normcase` function does not accurately reflect the platform's case-sensitivity, so there is always the possibility of two apparently-different paths being equal on such platforms.

## History

### 0.6c9

- Fix `resource_listdir('')` always returning an empty list for zipped eggs.

### 0.6c7

- Fix package precedence problem where single-version eggs installed in `site-packages` would take precedence over `.egg` files (or directories) installed in `site-packages`.

### 0.6c6

- Fix extracted C extensions not having executable permissions under Cygwin.
- Allow `.egg-link` files to contain relative paths.
- Fix cache dir defaults on Windows when multiple environment vars are needed to construct a path.

### 0.6c4

- Fix "dev" versions being considered newer than release candidates.

### 0.6c3

- Python 2.5 compatibility fixes.

### 0.6c2

- Fix a problem with eggs specified directly on `PYTHONPATH` on case-insensitive filesystems possibly not showing up in the default working set, due to differing normalizations of `sys.path` entries.

**0.6b3**

- Fixed a duplicate path insertion problem on case-insensitive filesystems.

**0.6b1**

- Split `get_platform()` into `get_supported_platform()` and `get_build_platform()` to work around a Mac versioning problem that caused the behavior of `compatible_platforms()` to be platform specific.
- Fix entry point parsing when a standalone module name has whitespace between it and the extras.

**0.6a11**

- Added `ExtractionError` and `ResourceManager.extraction_error()` so that cache permission problems get a more user-friendly explanation of the problem, and so that programs can catch and handle extraction errors if they need to.

**0.6a10**

- Added the `extras` attribute to `Distribution`, the `find_plugins()` method to `WorkingSet`, and the `__add__()` and `__iadd__()` methods to `Environment`.
- `safe_name()` now allows dots in project names.
- There is a new `to_filename()` function that escapes project names and versions for safe use in constructing egg filenames from a `Distribution` object's metadata.
- Added `Distribution.clone()` method, and keyword argument support to other `Distribution` constructors.
- Added the `DEVELOP_DIST` precedence, and automatically assign it to eggs using `.egg-info` format.

**0.6a9**

- Don't raise an error when an invalid (unfinished) distribution is found unless absolutely necessary. Warn about skipping invalid/unfinished eggs when building an `Environment`.
- Added support for `.egg-info` files or directories with version/platform information embedded in the filename, so that system packagers have the option of including `PKG-INFO` files to indicate the presence of a system-installed egg, without needing to use `.egg` directories, zipfiles, or `.pth` manipulation.
- Changed `parse_version()` to remove dashes before pre-release tags, so that `0.2-rc1` is considered an *older* version than `0.2`, and is equal to `0.2rc1`. The idea that a dash *always* meant a post-release version was highly non-intuitive to setuptools users and Python developers, who seem to want to use `-rc` version numbers a lot.

**0.6a8**

- Fixed a problem with `WorkingSet.resolve()` that prevented version conflicts from being detected at runtime.
- Improved runtime conflict warning message to identify a line in the user's program, rather than flagging the `warn()` call in `pkg_resources`.
- Avoid giving runtime conflict warnings for namespace packages, even if they were declared by a different package than the one currently being activated.
- Fix path insertion algorithm for case-insensitive filesystems.
- Fixed a problem with nested namespace packages (e.g. `peak.util`) not being set as an attribute of their parent package.

#### 0.6a6

- Activated distributions are now inserted in `sys.path` (and the working set) just before the directory that contains them, instead of at the end. This allows e.g. eggs in `site-packages` to override unmanaged modules in the same location, and allows eggs found earlier on `sys.path` to override ones found later.
- When a distribution is activated, it now checks whether any contained non-namespace modules have already been imported and issues a warning if a conflicting module has already been imported.
- Changed dependency processing so that it's breadth-first, allowing a depender's preferences to override those of a dependee, to prevent conflicts when a lower version is acceptable to the dependee, but not the depender.
- Fixed a problem extracting zipped files on Windows, when the egg in question has had changed contents but still has the same version number.

#### 0.6a4

- Fix a bug in `WorkingSet.resolve()` that was introduced in 0.6a3.

#### 0.6a3

- Added `safe_extra()` parsing utility routine, and use it for `Requirement`, `EntryPoint`, and `Distribution` objects' extras handling.

#### 0.6a1

- Enhanced performance of `require()` and related operations when all requirements are already in the working set, and enhanced performance of directory scanning for distributions.
- Fixed some problems using `pkg_resources` w/PEP 302 loaders other than `zipimport`, and the previously-broken "eager resource" support.
- Fixed `pkg_resources.resource_exists()` not working correctly, along with some other resource API bugs.
- Many API changes and enhancements:
  - Added `EntryPoint`, `get_entry_map`, `load_entry_point`, and `get_entry_info` APIs for dynamic plugin discovery.
  - `list_resources` is now `resource_listdir` (and it actually works)
  - Resource API functions like `resource_string()` that accepted a package name and resource name, will now also accept a `Requirement` object in place of the package name (to allow access to non-package data files in an egg).
  - `get_provider()` will now accept a `Requirement` instance or a module name. If it is given a `Requirement`, it will return a corresponding `Distribution` (by calling `require()` if a suitable distribution isn't already in the working set), rather than returning a metadata and resource provider for a specific module. (The difference is in how resource paths are interpreted; supplying a module name means resources path will be module-relative, rather than relative to the distribution's root.)
  - `Distribution` objects now implement the `IResourceProvider` and `IMetadataProvider` interfaces, so you don't need to reference the (no longer available) `metadata` attribute to get at these interfaces.
  - `Distribution` and `Requirement` both have a `project_name` attribute for the project name they refer to. (Previously these were `name` and `distname` attributes.)
  - The `path` attribute of `Distribution` objects is now `location`, because it isn't necessarily a filesystem path (and hasn't been for some time now). The `location` of

Distribution objects in the filesystem should always be normalized using `pkg_resources.normalize_path()`; all of the setuptools' code that generates distributions from the filesystem (including `Distribution.from_filename()`) ensure this invariant, but if you use a more generic API like `Distribution()` or `Distribution.from_location()` you should take care that you don't create a distribution with an un-normalized filesystem path.

- Distribution objects now have an `as_requirement()` method that returns a `Requirement` for the distribution's project name and version.
- Distribution objects no longer have an `installed_on()` method, and the `install_on()` method is now `activate()` (but may go away altogether soon). The `depends()` method has also been renamed to `requires()`, and `InvalidOption` is now `UnknownExtra`.
- `find_distributions()` now takes an additional argument called `only`, that tells it to only yield distributions whose location is the passed-in path. (It defaults to `False`, so that the default behavior is unchanged.)
- `AvailableDistributions` is now called `Environment`, and the `get()`, `__len__()`, and `__contains__()` methods were removed, because they weren't particularly useful. `__getitem__()` no longer raises `KeyError`; it just returns an empty list if there are no distributions for the named project.
- The `resolve()` method of `Environment` is now a method of `WorkingSet` instead, and the `best_match()` method now uses a working set instead of a path list as its second argument.
- There is a new `pkg_resources.add_activation_listener()` API that lets you register a callback for notifications about distributions added to `sys.path` (including the distributions already on it). This is basically a hook for extensible applications and frameworks to be able to search for plugin metadata in distributions added at runtime.

#### 0.5a13

- Fixed a bug in resource extraction from nested packages in a zipped egg.

#### 0.5a12

- Updated extraction/cache mechanism for zipped resources to avoid inter-process and inter-thread races during extraction. The default cache location can now be set via the `PYTHON_EGGS_CACHE` environment variable, and the default Windows cache is now a `Python-Eggs` subdirectory of the current user's "Application Data" directory, if the `PYTHON_EGGS_CACHE` variable isn't set.

#### 0.5a10

- Fix a problem with `pkg_resources` being confused by non-existent eggs on `sys.path` (e.g. if a user deletes an egg without removing it from the `easy-install.pth` file).
- Fix a problem with "basket" support in `pkg_resources`, where egg-finding never actually went inside .egg files.
- Made `pkg_resources` import the module you request resources from, if it's not already imported.

#### 0.5a4

- `pkg_resources.AvailableDistributions.resolve()` and related methods now accept an `installer` argument: a callable taking one argument, a `Requirement` instance. The callable must return a `Distribution` object, or `None` if no distribution is found. This feature is used by `EasyInstall` to resolve dependencies by recursively invoking itself.

#### 0.4a4

- Fix problems with `resource_listdir()`, `resource_isdir()` and resource directory extraction for zipped eggs.

#### 0.4a3

- Fixed scripts not being able to see a `__file__` variable in `__main__`
- Fixed a problem with `resource_isdir()` implementation that was introduced in 0.4a2.

#### 0.4a1

- Fixed a bug in requirements processing for exact versions (i.e. `==` and `!=`) when only one condition was included.
- Added `safe_name()` and `safe_version()` APIs to clean up handling of arbitrary distribution names and versions found on PyPI.

#### 0.3a4

- `pkg_resources` now supports resource directories, not just the resources in them. In particular, there are `resource_listdir()` and `resource_isdir()` APIs.
- `pkg_resources` now supports “egg baskets” – `.egg` zipfiles which contain multiple distributions in subdirectories whose names end with `.egg`. Having such a “basket” in a directory on `sys.path` is equivalent to having the individual eggs in that directory, but the contained eggs can be individually added (or not) to `sys.path`. Currently, however, there is no automated way to create baskets.
- Namespace package manipulation is now protected by the Python import lock.

#### 0.3a1

- Initial release.

---

# Supporting both Python 2 and Python 3 with Setuptools

---

Starting with Distribute version 0.6.2 and Setuptools 0.7, the Setuptools project supported Python 3. Installing and using setuptools for Python 3 code works exactly the same as for Python 2 code.

Setuptools provides a facility to invoke 2to3 on the code as a part of the build process, by setting the keyword parameter `use_2to3` to `True`, but the Setuptools project strongly recommends instead developing a unified codebase using `six`, `future`, or another compatibility library.

### 3.1 Using 2to3

Setuptools attempts to make the porting process easier by automatically running 2to3 as a part of running tests. To do so, you need to configure the `setup.py` so that you can run the unit tests with `python setup.py test`.

See *test - Build package and run a unittest suite* for more information on this.

Once you have the tests running under Python 2, you can add the `use_2to3` keyword parameters to `setup()`, and start running the tests under Python 3. The test command will now first run the build command during which the code will be converted with 2to3, and the tests will then be run from the build directory, as opposed from the source directory as is normally done.

Setuptools will convert all Python files, and also all doctests in Python files. However, if you have doctests located in separate text files, these will not automatically be converted. By adding them to the `convert_2to3_doctests` keyword parameter Setuptools will convert them as well.

By default, the conversion uses all fixers in the `lib2to3.fixers` package. To use additional fixers, the parameter `use_2to3_fixers` can be set to a list of names of packages containing fixers. To exclude fixers, the parameter `use_2to3_exclude_fixers` can be set to fixer names to be skipped.

An example `setup.py` might look something like this:

```
from setuptools import setup

setup(
    name='your.module',
```

(continues on next page)

(continued from previous page)

```
version='1.0',
description='This is your awesome module',
author='You',
author_email='your@email',
package_dir={'': 'src'},
packages=['your', 'you.module'],
test_suite='your.module.tests',
use_2to3=True,
convert_2to3_doctests=['src/your/module/README.txt'],
use_2to3_fixers=['your.fixers'],
use_2to3_exclude_fixers=['lib2to3.fixes.fix_import'],
)
```

### 3.1.1 Differential conversion

Note that a file will only be copied and converted during the build process if the source file has been changed. If you add a file to the doctests that should be converted, it will not be converted the next time you run the tests, since it hasn't been modified. You need to remove it from the build directory. Also if you run the build, install or test commands before adding the `use_2to3` parameter, you will have to remove the build directory before you run the test command, as the files otherwise will seem updated, and no conversion will happen.

In general, if code doesn't seem to be converted, deleting the build directory and trying again is a good safeguard against the build directory getting "out of sync" with the source directory.

## 3.2 Distributing Python 3 modules

You can distribute your modules with Python 3 support in different ways. A normal source distribution will work, but can be slow in installing, as the 2to3 process will be run during the install. But you can also distribute the module in binary format, such as a binary egg. That egg will contain the already converted code, and hence no 2to3 conversion is needed during install.

## 3.3 Advanced features

If you don't want to run the 2to3 conversion on the doctests in Python files, you can turn that off by setting `setuptools.use_2to3_on_doctests = False`.



---

## Development on Setuptools

---

Setuptools is maintained by the Python community under the Python Packaging Authority (PyPA) and led by Jason R. Coombs.

This document describes the process by which Setuptools is developed. This document assumes the reader has some passing familiarity with *using* setuptools, the `pkg_resources` module, and `pip`. It does not attempt to explain basic concepts like inter-project dependencies, nor does it contain detailed lexical syntax for most file formats. Neither does it explain concepts like “namespace packages” or “resources” in any detail, as all of these subjects are covered at length in the setuptools developer’s guide and the `pkg_resources` reference manual.

Instead, this is **internal** documentation for how those concepts and features are *implemented* in concrete terms. It is intended for people who are working on the setuptools code base, who want to be able to troubleshoot setuptools problems, want to write code that reads the file formats involved, or want to otherwise tinker with setuptools-generated files and directories.

Note, however, that these are all internal implementation details and are therefore subject to change; stick to the published API if you don’t want to be responsible for keeping your code from breaking when setuptools changes. You have been warned.

### 4.1 Developer’s Guide for Setuptools

If you want to know more about contributing on Setuptools, this is the place.

#### Table of Contents

- *Developer’s Guide for Setuptools*
  - *Recommended Reading*
  - *Project Management*
  - *Authoring Tickets*
  - *Making a pull request*

- *Auto-Merge Requests*
- *Testing*
- *Semantic Versioning*
- *Building Documentation*
- *Vendored Dependencies*

### 4.1.1 Recommended Reading

Please read [How to write the perfect pull request](#) for some tips on contributing to open source projects. Although the article is not authoritative, it was authored by the maintainer of Setuptools, so reflects his opinions and will improve the likelihood of acceptance and quality of contribution.

### 4.1.2 Project Management

Setuptools is maintained primarily in Github at [this home](#). Setuptools is maintained under the Python Packaging Authority (PyPA) with several core contributors. All bugs for Setuptools are filed and the canonical source is maintained in Github.

User support and discussions are done through the issue tracker (for specific) issues, through the distutils-sig mailing list, or on IRC (Freenode) at #pypa.

Discussions about development happen on the pypa-dev mailing list or on [Gitter](#).

### 4.1.3 Authoring Tickets

Before authoring any source code, it's often prudent to file a ticket describing the motivation behind making changes. First search to see if a ticket already exists for your issue. If not, create one. Try to think from the perspective of the reader. Explain what behavior you expected, what you got instead, and what factors might have contributed to the unexpected behavior. In Github, surround a block of code or traceback with the triple backtick “```” so that it is formatted nicely.

Filing a ticket provides a forum for justification, discussion, and clarification. The ticket provides a record of the purpose for the change and any hard decisions that were made. It provides a single place for others to reference when trying to understand why the software operates the way it does or why certain changes were made.

Setuptools makes extensive use of hyperlinks to tickets in the changelog so that system integrators and other users can get a quick summary, but then jump to the in-depth discussion about any subject referenced.

### 4.1.4 Making a pull request

When making a pull request, please include a short summary of the changes and a reference to any issue tickets that the PR is intended to solve. All PRs with code changes should include tests. All changes should include a changelog entry.

setuptools uses [towncrier](#) for changelog management, so when making a PR, please add a news fragment in the `changelog.d/` folder. Changelog files are written in reStructuredText and should be a 1 or 2 sentence description of the substantive changes in the PR. They should be named `<pr_number>.<category>.rst`, where the categories are:

- `change`: Any backwards compatible code change

- `breaking`: Any backwards-compatibility breaking change
- `doc`: A change to the documentation
- `misc`: Changes internal to the repo like CI, test and build changes
- `deprecation`: For deprecations of an existing feature or behavior

A pull request may have more than one of these components, for example a code change may introduce a new feature that deprecates an old feature, in which case two fragments should be added. It is not necessary to make a separate documentation fragment for documentation changes accompanying the relevant code changes. See the following for an example news fragment:

```
$ cat changelog.d/1288.change.rst
Add support for maintainer in PKG-INFO
```

### 4.1.5 Auto-Merge Requests

To support running all code through CI, even lightweight contributions, the project employs Mergify to auto-merge pull requests tagged as auto-merge.

Use `hub pull-request -l auto-merge` to create such a pull request from the command line after pushing a new branch.

### 4.1.6 Testing

The primary tests are run using tox. Make sure you have tox installed, and invoke it:

```
$ tox
```

Under continuous integration, additional tests may be run. See the `.travis.yml` file for full details on the tests run under Travis-CI.

### 4.1.7 Semantic Versioning

Setuptools follows `semver`.

### 4.1.8 Building Documentation

Setuptools relies on the `Sphinx` system for building documentation. The `published documentation` is hosted on Read the Docs.

To build the docs locally, use tox:

```
$ tox -e docs
```

### 4.1.9 Vendored Dependencies

Setuptools has some dependencies, but due to *bootstrapping issues* <<https://github.com/pypa/setuptools/issues/980>>, those dependencies cannot be declared as they won't be resolved soon enough to build setuptools from source. Eventually, this limitation may be lifted as PEP 517/518 reach ubiquitous adoption, but for now, Setuptools cannot declare dependencies other than through `setuptools/_vendor/vendored.txt` and `pkg_resources/_vendor/vendored.txt` and refreshed by way of `paver update_vendored` (`pavement.py`).

## 4.2 The Internal Structure of Python Eggs

STOP! This is not the first document you should read!

### Table of Contents

- *The Internal Structure of Python Eggs*
  - *Eggs and their Formats*
    - \* *Code and Resources*
    - \* *Project Metadata*
    - \* *Filename-Embedded Metadata*
    - \* *Egg Links*
  - *Standard Metadata*
    - \* *.txt File Formats*
    - \* *Dependency Metadata*
      - *requires.txt*
      - *setup\_requires.txt*
      - *dependency\_links.txt*
      - *depends.txt – Obsolete, do not create!*
    - \* *namespace\_packages.txt – Namespace Package Metadata*
    - \* *entry\_points.txt – “Entry Point”/Plugin Metadata*
    - \* *The scripts Subdirectory*
    - \* *Zip Support Metadata*
      - *native\_libs.txt*
      - *eager\_resources.txt*
      - *zip-safe and not-zip-safe*
    - \* *top\_level.txt – Conflict Management Metadata*
    - \* *SOURCES.txt – Source Files Manifest*
  - *Other Technical Considerations*
    - \* *Zip File Issues*
      - *The Extraction Process*
      - *Extension Import Wrappers*
    - \* *Installation and Path Management Issues*
      - *Script Wrappers*

### 4.2.1 Eggs and their Formats

A “Python egg” is a logical structure embodying the release of a specific version of a Python project, comprising its code, resources, and metadata. There are multiple formats that can be used to physically encode a Python egg, and others can be developed. However, a key principle of Python eggs is that they should be discoverable and importable. That is, it should be possible for a Python application to easily and efficiently find out what eggs are present on a system, and to ensure that the desired eggs’ contents are importable.

There are two basic formats currently implemented for Python eggs:

1. `.egg` format: a directory or zipfile *containing* the project’s code and resources, along with an `EGG-INFO` subdirectory that contains the project’s metadata
2. `.egg-info` format: a file or directory placed *adjacent* to the project’s code and resources, that directly contains the project’s metadata.

Both formats can include arbitrary Python code and resources, including static data files, package and non-package directories, Python modules, C extension modules, and so on. But each format is optimized for different purposes.

The `.egg` format is well-suited to distribution and the easy uninstallation or upgrades of code, since the project is essentially self-contained within a single directory or file, unmingled with any other projects’ code or resources. It also makes it possible to have multiple versions of a project simultaneously installed, such that individual programs can select the versions they wish to use.

The `.egg-info` format, on the other hand, was created to support backward-compatibility, performance, and ease of installation for system packaging tools that expect to install all projects’ code and resources to a single directory (e.g. `site-packages`). Placing the metadata in that same directory simplifies the installation process, since it isn’t necessary to create `.pth` files or otherwise modify `sys.path` to include each installed egg.

Its disadvantage, however, is that it provides no support for clean uninstallation or upgrades, and of course only a single version of a project can be installed to a given directory. Thus, support from a package management tool is required. (This is why setuptools’ “install” command refers to this type of egg installation as “single-version, externally managed”.) Also, they lack sufficient data to allow them to be copied from their installation source. `easy_install` can “ship” an application by copying `.egg` files or directories to a target location, but it cannot do this for `.egg-info` installs, because there is no way to tell what code and resources belong to a particular egg – there may be several eggs “scrambled” together in a single installation location, and the `.egg-info` format does not currently include a way to list the files that were installed. (This may change in a future version.)

#### Code and Resources

The layout of the code and resources is dictated by Python’s normal import layout, relative to the egg’s “base location”.

For the `.egg` format, the base location is the `.egg` itself. That is, adding the `.egg` filename or directory name to `sys.path` makes its contents importable.

For the `.egg-info` format, however, the base location is the directory that *contains* the `.egg-info`, and thus it is the directory that must be added to `sys.path` to make the egg importable. (Note that this means that the “normal” installation of a package to a `sys.path` directory is sufficient to make it an “egg” if it has an `.egg-info` file or directory installed alongside of it.)

#### Project Metadata

If eggs contained only code and resources, there would of course be no difference between them and any other directory or zip file on `sys.path`. Thus, metadata must also be included, using a metadata file or directory.

For the `.egg` format, the metadata is placed in an `EGG-INFO` subdirectory, directly within the `.egg` file or directory. For the `.egg-info` format, metadata is stored directly within the `.egg-info` directory itself.

The minimum project metadata that all eggs must have is a standard Python `PKG-INFO` file, named `PKG-INFO` and placed within the metadata directory appropriate to the format. Because it's possible for this to be the only metadata file included, `.egg-info` format eggs are not required to be a directory; they can just be a `.egg-info` file that directly contains the `PKG-INFO` metadata. This eliminates the need to create a directory just to store one file. This option is *not* available for `.egg` formats, since setuptools always includes other metadata. (In fact, setuptools itself never generates `.egg-info` files, either; the support for using files was added so that the requirement could easily be satisfied by other tools, such as distutils).

In addition to the `PKG-INFO` file, an egg's metadata directory may also include files and directories representing various forms of optional standard metadata (see the section on [Standard Metadata](#), below) or user-defined metadata required by the project. For example, some projects may define a metadata format to describe their application plugins, and metadata in this format would then be included by plugin creators in their projects' metadata directories.

## Filename-Embedded Metadata

To allow introspection of installed projects and runtime resolution of inter-project dependencies, a certain amount of information is embedded in egg filenames. At a minimum, this includes the project name, and ideally will also include the project version number. Optionally, it can also include the target Python version and required runtime platform if platform-specific C code is included. The syntax of an egg filename is as follows:

```
name ["-" version ["-py" pyver ["-" required_platform]]] "." ext
```

The “name” and “version” should be escaped using the `to_filename()` function provided by `pkg_resources`, after first processing them with `safe_name()` and `safe_version()` respectively. These latter two functions can also be used to later “unescape” these parts of the filename. (For a detailed description of these transformations, please see the “Parsing Utilities” section of the `pkg_resources` manual.)

The “pyver” string is the Python major version, as found in the first 3 characters of `sys.version`. “required\_platform” is essentially a `distutils.get_platform()` string, but with enhancements to properly distinguish Mac OS versions. (See the `get_build_platform()` documentation in the “Platform Utilities” section of the `pkg_resources` manual for more details.)

Finally, the “ext” is either `.egg` or `.egg-info`, as appropriate for the egg's format.

Normally, an egg's filename should include at least the project name and version, as this allows the runtime system to find desired project versions without having to read the egg's `PKG-INFO` to determine its version number.

Setuptools, however, only includes the version number in the filename when an `.egg` file is built using the `bdist_egg` command, or when an `.egg-info` directory is being installed by the `install_egg_info` command. When generating metadata for use with the original source tree, it only includes the project name, so that the directory will not have to be renamed each time the project's version changes.

This is especially important when version numbers change frequently, and the source metadata directory is kept under version control with the rest of the project. (As would be the case when the project's source includes project-defined metadata that is not generated from by setuptools from data in the setup script.)

## Egg Links

In addition to the `.egg` and `.egg-info` formats, there is a third egg-related extension that you may encounter on occasion: `.egg-link` files.

These files are not eggs, strictly speaking. They simply provide a way to reference an egg that is not physically installed in the desired location. They exist primarily as a cross-platform alternative to symbolic links, to support “installing” code that is being developed in a different location than the desired installation location. For example, if a user is developing an application plugin in their home directory, but the plugin needs to be “installed” in an application plugin

directory, running “`setup.py develop -md /path/to/app/plugins`” will install an `.egg-link` file in `/path/to/app/plugins`, that tells the egg runtime system where to find the actual egg (the user’s project source directory and its `.egg-info` subdirectory).

`.egg-link` files are named following the format for `.egg` and `.egg-info` names, but only the project name is included; no version, Python version, or platform information is included. When the runtime searches for available eggs, `.egg-link` files are opened and the actual egg file/directory name is read from them.

Each `.egg-link` file should contain a single file or directory name, with no newlines. This filename should be the base location of one or more eggs. That is, the name must either end in `.egg`, or else it should be the parent directory of one or more `.egg-info` format eggs.

As of setuptools 0.6c6, the path may be specified as a platform-independent (i.e. `/`-separated) relative path from the directory containing the `.egg-link` file, and a second line may appear in the file, specifying a platform-independent relative path from the egg’s base directory to its setup script directory. This allows installation tools such as EasyInstall to find the project’s setup directory and build eggs or perform other setup commands on it.

## 4.2.2 Standard Metadata

In addition to the minimum required `PKG-INFO` metadata, projects can include a variety of standard metadata files or directories, as described below. Except as otherwise noted, these files and directories are automatically generated by setuptools, based on information supplied in the setup script or through analysis of the project’s code and resources.

Most of these files and directories are generated via “egg-info writers” during execution of the setuptools `egg_info` command, and are listed in the `egg_info.writers` entry point group defined by setuptools’ own `setup.py` file.

Project authors can register their own metadata writers as entry points in this group (as described in the setuptools manual under “Adding new EGG-INFO Files”) to cause setuptools to generate project-specific metadata files or directories during execution of the `egg_info` command. It is up to project authors to document these new metadata formats, if they create any.

### .txt File Formats

Files described in this section that have `.txt` extensions have a simple lexical format consisting of a sequence of text lines, each line terminated by a linefeed character (regardless of platform). Leading and trailing whitespace on each line is ignored, as are blank lines and lines whose first nonblank character is a `#` (comment symbol). (This is the parsing format defined by the `yield_lines()` function of the `pkg_resources` module.)

All `.txt` files defined by this section follow this format, but some are also “sectioned” files, meaning that their contents are divided into sections, using square-bracketed section headers akin to Windows `.ini` format. Note that this does *not* imply that the lines within the sections follow an `.ini` format, however. Please see an individual metadata file’s documentation for a description of what the lines and section names mean in that particular file.

Sectioned files can be parsed using the `split_sections()` function; see the “Parsing Utilities” section of the `pkg_resources` manual for details.

## Dependency Metadata

### `requires.txt`

This is a “sectioned” text file. Each section is a sequence of “requirements”, as parsed by the `parse_requirements()` function; please see the `pkg_resources` manual for the complete requirement parsing syntax.

The first, unnamed section (i.e., before the first section header) in this file is the project’s core requirements, which must be installed for the project to function. (Specified using the `install_requires` keyword to `setup()`).

The remaining (named) sections describe the project’s “extra” requirements, as specified using the `extras_require` keyword to `setup()`. The section name is the name of the optional feature, and the section body lists that feature’s dependencies.

Note that it is not normally necessary to inspect this file directly; `pkg_resources.Distribution` objects have a `requires()` method that can be used to obtain `Requirement` objects describing the project’s core and optional dependencies.

### `setup_requires.txt`

Much like `requires.txt` except represents the requirements specified by the `setup_requires` parameter to the `Distribution`.

### `dependency_links.txt`

A list of dependency URLs, one per line, as specified using the `dependency_links` keyword to `setup()`. These may be direct download URLs, or the URLs of web pages containing direct download links. Please see the setuptools manual for more information on specifying this option.

### `depends.txt` – Obsolete, do not create!

This file follows an identical format to `requires.txt`, but is obsolete and should not be used. The earliest versions of setuptools required users to manually create and maintain this file, so the runtime still supports reading it, if it exists. The new filename was created so that it could be automatically generated from `setup()` information without overwriting an existing hand-created `depends.txt`, if one was already present in the project’s source `.egg-info` directory.

### `namespace_packages.txt` – Namespace Package Metadata

A list of namespace package names, one per line, as supplied to the `namespace_packages` keyword to `setup()`. Please see the manuals for setuptools and `pkg_resources` for more information about namespace packages.

### `entry_points.txt` – “Entry Point”/Plugin Metadata

This is a “sectioned” text file, whose contents encode the `entry_points` keyword supplied to `setup()`. All sections are named, as the section names specify the entry point groups in which the corresponding section’s entry points are registered.

Each section is a sequence of “entry point” lines, each parseable using the `EntryPoint.parse` classmethod; please see the `pkg_resources` manual for the complete entry point parsing syntax.

Note that it is not necessary to parse this file directly; the `pkg_resources` module provides a variety of APIs to locate and load entry points automatically. Please see the setuptools and `pkg_resources` manuals for details on the nature and uses of entry points.



## The `scripts` Subdirectory

This directory is currently only created for `.egg` files built by the `setuptools bdist_egg` command. It will contain copies of all of the project’s “traditional” scripts (i.e., those specified using the `scripts` keyword to `setup()`). This is so that they can be reconstituted when an `.egg` file is installed.

The scripts are placed here using the `distutils`’ standard `install_scripts` command, so any `#!` lines reflect the Python installation where the egg was built. But instead of copying the scripts to the local script installation directory, EasyInstall writes short wrapper scripts that invoke the original scripts from inside the egg, after ensuring that `sys.path` includes the egg and any eggs it depends on. For more about *script wrappers*, see the section below on *Installation and Path Management Issues*.

## Zip Support Metadata

### `native_libs.txt`

A list of C extensions and other dynamic link libraries contained in the egg, one per line. Paths are `/`-separated and relative to the egg’s base location.

This file is generated as part of `bdist_egg` processing, and as such only appears in `.egg` files (and `.egg` directories created by unpacking them). It is used to ensure that all libraries are extracted from a zipped egg at the same time, in case there is any direct linkage between them. Please see the *Zip File Issues* section below for more information on library and resource extraction from `.egg` files.

### `eager_resources.txt`

A list of resource files and/or directories, one per line, as specified via the `eager_resources` keyword to `setup()`. Paths are `/`-separated and relative to the egg’s base location.

Resource files or directories listed here will be extracted simultaneously, if any of the named resources are extracted, or if any native libraries listed in `native_libs.txt` are extracted. Please see the `setuptools` manual for details on what this feature is used for and how it works, as well as the *Zip File Issues* section below.

### `zip-safe` and `not-zip-safe`

These are zero-length files, and either one or the other should exist. If `zip-safe` exists, it means that the project will work properly when installed as an `.egg` zipfile, and conversely the existence of `not-zip-safe` means the project should not be installed as an `.egg` file. The `zip_safe` option to `setuptools`’ `setup()` determines which file will be written. If the option isn’t provided, `setuptools` attempts to make its own assessment of whether the package can work, based on code and content analysis.

If neither file is present at installation time, EasyInstall defaults to assuming that the project should be unzipped. (Command-line options to EasyInstall, however, take precedence even over an existing `zip-safe` or `not-zip-safe` file.)

Note that these flag files appear only in `.egg` files generated by `bdist_egg`, and in `.egg` directories created by unpacking such an `.egg` file.

### `top_level.txt` – Conflict Management Metadata

This file is a list of the top-level module or package names provided by the project, one Python identifier per line.

Subpackages are not included; a project containing both a `foo.bar` and a `foo.baz` would include only one line, `foo`, in its `top_level.txt`.

This data is used by `pkg_resources` at runtime to issue a warning if an egg is added to `sys.path` when its contained packages may have already been imported.

(It was also once used to detect conflicts with non-egg packages at installation time, but in more recent versions, setuptools installs eggs in such a way that they always override non-egg packages, thus preventing a problem from arising.)

### **SOURCES.txt – Source Files Manifest**

This file is roughly equivalent to the distutils’ `MANIFEST` file. The differences are as follows:

- The filenames always use `/` as a path separator, which must be converted back to a platform-specific path whenever they are read.
- The file is automatically generated by setuptools whenever the `egg_info` or `sdist` commands are run, and it is *not* user-editable.

Although this metadata is included with distributed eggs, it is not actually used at runtime for any purpose. Its function is to ensure that setuptools-built *source* distributions can correctly discover what files are part of the project’s source, even if the list had been generated using revision control metadata on the original author’s system.

In other words, `SOURCES.txt` has little or no runtime value for being included in distributed eggs, and it is possible that future versions of the `bdist_egg` and `install_egg_info` commands will strip it before installation or distribution. Therefore, do not rely on its being available outside of an original source directory or source distribution.

## **4.2.3 Other Technical Considerations**

### **Zip File Issues**

Although zip files resemble directories, they are not fully substitutable for them. Most platforms do not support loading dynamic link libraries contained in zipfiles, so it is not possible to directly import C extensions from `.egg` zipfiles. Similarly, there are many existing libraries – whether in Python or C – that require actual operating system filenames, and do not work with arbitrary “file-like” objects or in-memory strings, and thus cannot operate directly on the contents of zip files.

To address these issues, the `pkg_resources` module provides a “resource API” to support obtaining either the contents of a resource, or a true operating system filename for the resource. If the egg containing the resource is a directory, the resource’s real filename is simply returned. However, if the egg is a zipfile, then the resource is first extracted to a cache directory, and the filename within the cache is returned.

The cache directory is determined by the `pkg_resources` API; please see the `set_cache_path()` and `get_default_cache()` documentation for details.

### **The Extraction Process**

Resources are extracted to a cache subdirectory whose name is based on the enclosing `.egg` filename and the path to the resource. If there is already a file of the correct name, size, and timestamp, its filename is returned to the requester. Otherwise, the desired file is extracted first to a temporary name generated using `mkstemp("%$extract", target_dir)`, and then its timestamp is set to match the one in the zip file, before renaming it to its final name. (Some collision detection and resolution code is used to handle the fact that Windows doesn’t overwrite files when renaming.)

If a resource directory is requested, all of its contents are recursively extracted in this fashion, to ensure that the directory name can be used as if it were valid all along.

If the resource requested for extraction is listed in the `native_libs.txt` or `eager_resources.txt` metadata files, then *all* resources listed in *either* file will be extracted before the requested resource's filename is returned, thus ensuring that all C extensions and data used by them will be simultaneously available.

## Extension Import Wrappers

Since Python's built-in zip import feature does not support loading C extension modules from zipfiles, the `setuptools` `bdist_egg` command generates special import wrappers to make it work.

The wrappers are `.py` files (along with corresponding `.pyc` and/or `.pyo` files) that have the same module name as the corresponding C extension. These wrappers are located in the same package directory (or top-level directory) within the zipfile, so that say, `foomodule.so` will get a corresponding `foo.py`, while `bar/baz.pyd` will get a corresponding `bar/baz.py`.

These wrapper files contain a short stanza of Python code that asks `pkg_resources` for the filename of the corresponding C extension, then reloads the module using the obtained filename. This will cause `pkg_resources` to first ensure that all of the egg's C extensions (and any accompanying "eager resources") are extracted to the cache before attempting to link to the C library.

Note, by the way, that `.egg` directories will also contain these wrapper files. However, Python's default import priority is such that C extensions take precedence over same-named Python modules, so the import wrappers are ignored unless the egg is a zipfile.

## Installation and Path Management Issues

Python's initial setup of `sys.path` is very dependent on the Python version and installation platform, as well as how Python was started (i.e., `script` vs. `-c` vs. `-m` vs. interactive interpreter). In fact, Python also provides only two relatively robust ways to affect `sys.path` outside of direct manipulation in code: the `PYTHONPATH` environment variable, and `.pth` files.

However, with no cross-platform way to safely and persistently change environment variables, this leaves `.pth` files as EasyInstall's only real option for persistent configuration of `sys.path`.

But `.pth` files are rather strictly limited in what they are allowed to do normally. They add directories only to the *end* of `sys.path`, after any locally-installed `site-packages` directory, and they are only processed *in* the `site-packages` directory to start with.

This is a double whammy for users who lack write access to that directory, because they can't create a `.pth` file that Python will read, and even if a sympathetic system administrator adds one for them that calls `site.addsitedir()` to allow some other directory to contain `.pth` files, they won't be able to install newer versions of anything that's installed in the systemwide `site-packages`, because their paths will still be added *after* `site-packages`.

So EasyInstall applies two workarounds to solve these problems.

The first is that EasyInstall leverages `.pth` files' "import" feature to manipulate `sys.path` and ensure that anything EasyInstall adds to a `.pth` file will always appear before both the standard library and the local `site-packages` directories. Thus, it is always possible for a user who can write a Python-readable `.pth` file to ensure that their packages come first in their own environment.

Second, when installing to a `PYTHONPATH` directory (as opposed to a "site" directory like `site-packages`) EasyInstall will also install a special version of the `site` module. Because it's in a `PYTHONPATH` directory, this module will get control before the standard library version of `site` does. It will record the state of `sys.path` before invoking the "real" `site` module, and then afterwards it processes any `.pth` files found in `PYTHONPATH` directories,

including all the fixups needed to ensure that eggs always appear before the standard library in `sys.path`, but are in a relative order to one another that is defined by their `PYTHONPATH` and `.pth`-prescribed sequence.

The net result of these changes is that `sys.path` order will be as follows at runtime:

1. The `sys.argv[0]` directory, or an empty string if no script is being executed.
2. All eggs installed by EasyInstall in any `.pth` file in each `PYTHONPATH` directory, in order first by `PYTHONPATH` order, then normal `.pth` processing order (which is to say alphabetical by `.pth` filename, then by the order of listing within each `.pth` file).
3. All eggs installed by EasyInstall in any `.pth` file in each “site” directory (such as `site-packages`), following the same ordering rules as for the ones on `PYTHONPATH`.
4. The `PYTHONPATH` directories themselves, in their original order
5. Any paths from `.pth` files found on `PYTHONPATH` that were *not* eggs installed by EasyInstall, again following the same relative ordering rules.
6. The standard library and “site” directories, along with the contents of any `.pth` files found in the “site” directories.

Notice that sections 1, 4, and 6 comprise the “normal” Python setup for `sys.path`. Sections 2 and 3 are inserted to support eggs, and section 5 emulates what the “normal” semantics of `.pth` files on `PYTHONPATH` would be if Python natively supported them.

For further discussion of the tradeoffs that went into this design, as well as notes on the actual magic inserted into `.pth` files to make them do these things, please see also the following messages to the distutils-SIG mailing list:

- <http://mail.python.org/pipermail/distutils-sig/2006-February/006026.html>
- <http://mail.python.org/pipermail/distutils-sig/2006-March/006123.html>

## Script Wrappers

EasyInstall never directly installs a project’s original scripts to a script installation directory. Instead, it writes short wrapper scripts that first ensure that the project’s dependencies are active on `sys.path`, before invoking the original script. These wrappers have a `#!` line that points to the version of Python that was used to install them, and their second line is always a comment that indicates the type of script wrapper, the project version required for the script to run, and information identifying the script to be invoked.

The format of this marker line is:

```
"# EASY-INSTALL-" script_type ": " tuple_of_strings "\n"
```

The `script_type` is one of `SCRIPT`, `DEV-SCRIPT`, or `ENTRY-SCRIPT`. The `tuple_of_strings` is a comma-separated sequence of Python string constants. For `SCRIPT` and `DEV-SCRIPT` wrappers, there are two strings: the project version requirement, and the script name (as a filename within the `scripts` metadata directory). For `ENTRY-SCRIPT` wrappers, there are three: the project version requirement, the entry point group name, and the entry point name. (See the “Automatic Script Creation” section in the setuptools manual for more information about entry point scripts.)

In each case, the project version requirement string will be a string parseable with the `pkg_resources` modules’ `Requirement.parse()` classmethod. The only difference between a `SCRIPT` wrapper and a `DEV-SCRIPT` is that a `DEV-SCRIPT` actually executes the original source script in the project’s source tree, and is created when the “`setup.py develop`” command is run. A `SCRIPT` wrapper, on the other hand, uses the “installed” script written to the `EGG-INFO/scripts` subdirectory of the corresponding `.egg` zipfile or directory. (`.egg-info` eggs do not have script wrappers associated with them, except in the “`setup.py develop`” case.)

The purpose of including the marker line in generated script wrappers is to facilitate introspection of installed scripts, and their relationship to installed eggs. For example, an uninstallation tool could use this data to identify what scripts can safely be removed, and/or identify what scripts would stop working if a particular egg is uninstalled.

## 4.3 Release Process

In order to allow for rapid, predictable releases, Setuptools uses a mechanical technique for releases, enacted on tagged commits by continuous integration.

To finalize a release, run `tox -e finalize`, review, then push the changes.

If tests pass, the release will be uploaded to PyPI.

### 4.3.1 Release Frequency

Some have asked why Setuptools is released so frequently. Because Setuptools uses a mechanical release process, it's very easy to make releases whenever the code is stable (tests are passing). As a result, the philosophy is to release early and often.

While some find the frequent releases somewhat surprising, they only empower the user. Although releases are made frequently, users can choose the frequency at which they use those releases. If instead Setuptools contributions were only released in batches, the user would be constrained to only use Setuptools when those official releases were made. With frequent releases, the user can govern exactly how often he wishes to update.

Frequent releases also then obviate the need for dev or beta releases in most cases. Because releases are made early and often, bugs are discovered and corrected quickly, in many cases before other users have yet to encounter them.

### 4.3.2 Release Managers

Additionally, anyone with push access to the master branch has access to cut releases.



## CHAPTER 5

---

### Roadmap

---

Setuptools maintains a series of [milestones](#) to track a roadmap of large-scale goals.





**Warning:** Easy Install is deprecated. Do not use it. Instead use pip. If you think you need Easy Install, please reach out to the PyPA team (a ticket to pip or setuptools is fine), describing your use-case.

Easy Install is a python module (`easy_install`) bundled with `setuptools` that lets you automatically download, build, install, and manage Python packages.

Please share your experiences with us! If you encounter difficulty installing a package, please contact us via the [distutils mailing list](#). (Note: please DO NOT send private email directly to the author of `setuptools`; it will be discarded. The mailing list is a searchable archive of previously-asked and answered questions; you should begin your research there before reporting something as a bug – and then do so via list discussion first.)

(Also, if you'd like to learn about how you can use `setuptools` to make your own packages work better with EasyInstall, or provide EasyInstall-like features without requiring your users to use EasyInstall directly, you'll probably want to check out the full documentation as well.)

### Table of Contents

- *Easy Install*
  - *Using “Easy Install”*
    - \* *Installing “Easy Install”*
      - *Troubleshooting*
      - *Windows Notes*
    - \* *Downloading and Installing a Package*
    - \* *Upgrading a Package*
    - \* *Changing the Active Version*
    - \* *Uninstalling Packages*

- \* *Managing Scripts*
- \* *Executables and Launchers*
  - *Windows Executable Launcher*
  - *Natural Script Launcher*
- \* *Tips & Techniques*
  - *Multiple Python Versions*
  - *Restricting Downloads with `--allow-hosts`*
  - *Installing on Un-networked Machines*
  - *Packaging Others' Projects As Eggs*
  - *Creating your own Package Index*
- \* *Password-Protected Sites*
- \* *Using .pypirc Credentials*
  - *Controlling Build Options*
  - *Editing and Viewing Source Packages*
  - *Dealing with Installation Conflicts*
  - *Compressed Installation*
- *Reference Manual*
  - \* *Configuration Files*
  - \* *Command-Line Options*
  - \* *Custom Installation Locations*
    - *Use the “`-user`” option*
    - *Use the “`-user`” option and customize “`PYTHONUSERBASE`”*
    - *Use “`virtualenv`”*
  - \* *Package Index “API”*

## 6.1 Using “Easy Install”

### 6.1.1 Installing “Easy Install”

Please see the [setuptools PyPI page](#) for download links and basic installation instructions for each of the supported platforms.

You will need at least Python 3.5 or 2.7. An `easy_install` script will be installed in the normal location for Python scripts on your platform.

Note that the instructions on the setuptools PyPI page assume that you are installing to Python’s primary `site-packages` directory. If this is not the case, you should consult the section below on [Custom Installation Locations](#) before installing. (And, on Windows, you should not use the `.exe` installer when installing to an alternate location.)

Note that `easy_install` normally works by downloading files from the internet. If you are behind an NTLM-based

firewall that prevents Python programs from accessing the net directly, you may wish to first install and use the [APS proxy server](#), which lets you get past such firewalls in the same way that your web browser(s) do.

(Alternately, if you do not wish `easy_install` to actually download anything, you can restrict it from doing so with the `--allow-hosts` option; see the sections on [restricting downloads with `--allow-hosts`](#) and [command-line options](#) for more details.)

## Troubleshooting

If EasyInstall/setuptools appears to install correctly, and you can run the `easy_install` command but it fails with an `ImportError`, the most likely cause is that you installed to a location other than `site-packages`, without taking any of the steps described in the [Custom Installation Locations](#) section below. Please see that section and follow the steps to make sure that your custom location will work correctly. Then re-install.

Similarly, if you can run `easy_install`, and it appears to be installing packages, but then you can't import them, the most likely issue is that you installed EasyInstall correctly but are using it to install packages to a non-standard location that hasn't been properly prepared. Again, see the section on [Custom Installation Locations](#) for more details.

## Windows Notes

Installing setuptools will provide an `easy_install` command according to the techniques described in [Executables and Launchers](#). If the `easy_install` command is not available after installation, that section provides details on how to configure Windows to make the commands available.

### 6.1.2 Downloading and Installing a Package

For basic use of `easy_install`, you need only supply the filename or URL of a source distribution or .egg file ([Python Egg](#)).

**Example 1.** Install a package by name, searching PyPI for the latest version, and automatically downloading, building, and installing it:

```
easy_install SQLAlchemy
```

**Example 2.** Install or upgrade a package by name and version by finding links on a given “download page”:

```
easy_install -f http://pythonpaste.org/package_index.html SQLAlchemy
```

**Example 3.** Download a source distribution from a specified URL, automatically building and installing it:

```
easy_install http://example.com/path/to/MyPackage-1.2.3.tgz
```

**Example 4.** Install an already-downloaded .egg file:

```
easy_install /my_downloads/OtherPackage-3.2.1-py2.3.egg
```

**Example 5.** Upgrade an already-installed package to the latest version listed on PyPI:

```
easy_install --upgrade PyProtocols
```

**Example 6.** Install a source distribution that's already downloaded and extracted in the current directory (New in 0.5a9):

```
easy_install .
```

**Example 7.** (New in 0.6a1) Find a source distribution or Subversion checkout URL for a package, and extract it or check it out to `~/projects/sqlobject` (the name will always be in all-lowercase), where it can be examined or edited. (The package will not be installed, but it can easily be installed with `easy_install ~/projects/sqlobject`. See *Editing and Viewing Source Packages* below for more info.):

```
easy_install --editable --build-directory ~/projects SQLAlchemy
```

**Example 7.** (New in 0.6.11) Install a distribution within your home dir:

```
easy_install --user SQLAlchemy
```

Easy Install accepts URLs, filenames, PyPI package names (i.e., distutils “distribution” names), and package+version specifiers. In each case, it will attempt to locate the latest available version that meets your criteria.

When downloading or processing downloaded files, Easy Install recognizes distutils source distribution files with extensions of `.tgz`, `.tar`, `.tar.gz`, `.tar.bz2`, or `.zip`. And of course it handles already-built `.egg` distributions as well as `.win32.exe` installers built using distutils.

By default, packages are installed to the running Python installation’s `site-packages` directory, unless you provide the `-d` or `--install-dir` option to specify an alternative directory, or specify an alternate location using distutils configuration files. (See *Configuration Files*, below.)

By default, any scripts included with the package are installed to the running Python installation’s standard script installation location. However, if you specify an installation directory via the command line or a config file, then the default directory for installing scripts will be the same as the package installation directory, to ensure that the script will have access to the installed package. You can override this using the `-s` or `--script-dir` option.

Installed packages are added to an `easy-install.pth` file in the install directory, so that Python will always use the most-recently-installed version of the package. If you would like to be able to select which version to use at runtime, you should use the `-m` or `--multi-version` option.

### 6.1.3 Upgrading a Package

You don’t need to do anything special to upgrade a package: just install the new version, either by requesting a specific version, e.g.:

```
easy_install "SomePackage==2.0"
```

a version greater than the one you have now:

```
easy_install "SomePackage>2.0"
```

using the upgrade flag, to find the latest available version on PyPI:

```
easy_install --upgrade SomePackage
```

or by using a download page, direct download URL, or package filename:

```
easy_install -f http://example.com/downloads ExamplePackage
easy_install http://example.com/downloads/ExamplePackage-2.0-py2.4.egg
easy_install my_downloads/ExamplePackage-2.0.tgz
```

If you’re using `-m` or `--multi-version`, using the `require()` function at runtime automatically selects the newest installed version of a package that meets your version criteria. So, installing a newer version is the only step needed to upgrade such packages.

If you're installing to a directory on PYTHONPATH, or a configured "site" directory (and not using `-m`), installing a package automatically replaces any previous version in the `easy-install.pth` file, so that Python will import the most-recently installed version by default. So, again, installing the newer version is the only upgrade step needed.

If you haven't suppressed script installation (using `--exclude-scripts` or `-x`), then the upgraded version's scripts will be installed, and they will be automatically patched to `require()` the corresponding version of the package, so that you can use them even if they are installed in multi-version mode.

`easy_install` never actually deletes packages (unless you're installing a package with the same name and version number as an existing package), so if you want to get rid of older versions of a package, please see [Uninstalling Packages](#), below.

## 6.1.4 Changing the Active Version

If you've upgraded a package, but need to revert to a previously-installed version, you can do so like this:

```
easy_install PackageName==1.2.3
```

Where `1.2.3` is replaced by the exact version number you wish to switch to. If a package matching the requested name and version is not already installed in a directory on `sys.path`, it will be located via PyPI and installed.

If you'd like to switch to the latest installed version of `PackageName`, you can do so like this:

```
easy_install PackageName
```

This will activate the latest installed version. (Note: if you have set any `find_links` via distutils configuration files, those download pages will be checked for the latest available version of the package, and it will be downloaded and installed if it is newer than your current version.)

Note that changing the active version of a package will install the newly active version's scripts, unless the `--exclude-scripts` or `-x` option is specified.

## 6.1.5 Uninstalling Packages

If you have replaced a package with another version, then you can just delete the package(s) you don't need by deleting the `PackageName-versioninfo.egg` file or directory (found in the installation directory).

If you want to delete the currently installed version of a package (or all versions of a package), you should first run:

```
easy_install -m PackageName
```

This will ensure that Python doesn't continue to search for a package you're planning to remove. After you've done this, you can safely delete the `.egg` files or directories, along with any scripts you wish to remove.

## 6.1.6 Managing Scripts

Whenever you install, upgrade, or change versions of a package, EasyInstall automatically installs the scripts for the selected package version, unless you tell it not to with `-x` or `--exclude-scripts`. If any scripts in the script directory have the same name, they are overwritten.

Thus, you do not normally need to manually delete scripts for older versions of a package, unless the newer version of the package does not include a script of the same name. However, if you are completely uninstalling a package, you may wish to manually delete its scripts.

EasyInstall's default behavior means that you can normally only run scripts from one version of a package at a time. If you want to keep multiple versions of a script available, however, you can simply use the `--multi-version`

or `-m` option, and rename the scripts that EasyInstall creates. This works because EasyInstall installs scripts as short code stubs that `require()` the matching version of the package the script came from, so renaming the script has no effect on what it executes.

For example, suppose you want to use two versions of the `rst2html` tool provided by the `docutils` package. You might first install one version:

```
easy_install -m docutils==0.3.9
```

then rename the `rst2html.py` to `r2h_039`, and install another version:

```
easy_install -m docutils==0.3.10
```

This will create another `rst2html.py` script, this one using `docutils` version 0.3.10 instead of 0.3.9. You now have two scripts, each using a different version of the package. (Notice that we used `-m` for both installations, so that Python won't lock us out of using anything but the most recently-installed version of the package.)

## 6.1.7 Executables and Launchers

On Unix systems, scripts are installed with as natural files with a “#!” header and no extension and they launch under the Python version indicated in the header.

On Windows, there is no mechanism to “execute” files without extensions, so EasyInstall provides two techniques to mirror the Unix behavior. The behavior is indicated by the `SETUPTOOLS_LAUNCHER` environment variable, which may be “executable” (default) or “natural”.

Regardless of the technique used, the script(s) will be installed to a Scripts directory (by default in the Python installation directory). It is recommended for EasyInstall that you ensure this directory is in the `PATH` environment variable. The easiest way to ensure the Scripts directory is in the `PATH` is to run `Tools\Scripts\win_add2path.py` from the Python directory.

Note that instead of changing your `PATH` to include the Python scripts directory, you can also retarget the installation location for scripts so they go on a directory that's already on the `PATH`. For more information see [Command-Line Options](#) and [Configuration Files](#). During installation, pass command line options (such as `--script-dir`) to control where scripts will be installed.

### Windows Executable Launcher

If the “executable” launcher is used, EasyInstall will create a ‘.exe’ launcher of the same name beside each installed script (including `easy_install` itself). These small .exe files launch the script of the same name using the Python version indicated in the “#!” header.

This behavior is currently default. To force the use of executable launchers, set `SETUPTOOLS_LAUNCHER` to “executable”.

### Natural Script Launcher

EasyInstall also supports deferring to an external launcher such as `pylauncher` for launching scripts. Enable this experimental functionality by setting the `SETUPTOOLS_LAUNCHER` environment variable to “natural”. EasyInstall will then install scripts as simple scripts with a .pya (or .pyw) extension appended. If these extensions are associated with the `pylauncher` and listed in the `PATHEXT` environment variable, these scripts can then be invoked simply and directly just like any other executable. This behavior may become default in a future version.

EasyInstall uses the .pya extension instead of simply the typical ‘.py’ extension. This distinct extension is necessary to prevent Python from treating the scripts as importable modules (where name conflicts exist). Current releases of `pylauncher` do not yet associate with .pya files by default, but future versions should do so.

## 6.1.8 Tips & Techniques

### Multiple Python Versions

EasyInstall installs itself under two names: `easy_install` and `easy_install-N.N`, where `N.N` is the Python version used to install it. Thus, if you install EasyInstall for both Python 3.2 and 2.7, you can use the `easy_install-3.2` or `easy_install-2.7` scripts to install packages for the respective Python version.

Setuptools also supplies `easy_install` as a runnable module which may be invoked using `python -m easy_install` for any Python with Setuptools installed.

### Restricting Downloads with `--allow-hosts`

You can use the `--allow-hosts (-H)` option to restrict what domains EasyInstall will look for links and downloads on. `--allow-hosts=None` prevents downloading altogether. You can also use wildcards, for example to restrict downloading to hosts in your own intranet. See the section below on *Command-Line Options* for more details on the `--allow-hosts` option.

By default, there are no host restrictions in effect, but you can change this default by editing the appropriate *configuration files* and adding:

```
[easy_install]
allow_hosts = *.myintranet.example.com,*.python.org
```

The above example would then allow downloads only from hosts in the `python.org` and `myintranet.example.com` domains, unless overridden on the command line.

### Installing on Un-networked Machines

Just copy the eggs or source packages you need to a directory on the target machine, then use the `-f` or `--find-links` option to specify that directory's location. For example:

```
easy_install -H None -f somedir SomePackage
```

will attempt to install `SomePackage` using only eggs and source packages found in `somedir` and disallowing all remote access. You should of course make sure you have all of `SomePackage`'s dependencies available in `somedir`.

If you have another machine of the same operating system and library versions (or if the packages aren't platform-specific), you can create the directory of eggs using a command like this:

```
easy_install -zmaxd somedir SomePackage
```

This will tell EasyInstall to put zipped eggs or source packages for `SomePackage` and all its dependencies into `somedir`, without creating any scripts or `.pth` files. You can then copy the contents of `somedir` to the target machine. (`-z` means zipped eggs, `-m` means multi-version, which prevents `.pth` files from being used, `-a` means to copy all the eggs needed, even if they're installed elsewhere on the machine, and `-d` indicates the directory to place the eggs in.)

You can also build the eggs from local development packages that were installed with the `setup.py develop` command, by including the `-l` option, e.g.:

```
easy_install -zmaxld somedir SomePackage
```

This will use locally-available source distributions to build the eggs.

## Packaging Others' Projects As Eggs

Need to distribute a package that isn't published in egg form? You can use EasyInstall to build eggs for a project. You'll want to use the `--zip-ok`, `--exclude-scripts`, and possibly `--no-deps` options (`-z`, `-x` and `-N`, respectively). Use `-d` or `--install-dir` to specify the location where you'd like the eggs placed. By placing them in a directory that is published to the web, you can then make the eggs available for download, either in an intranet or to the internet at large.

If someone distributes a package in the form of a single `.py` file, you can wrap it in an egg by tacking an `#egg=name-version` suffix on the file's URL. So, something like this:

```
easy_install -f "http://some.example.com/downloads/foo.py#egg=foo-1.0" foo
```

will install the package as an egg, and this:

```
easy_install -zmaxd. \
-f "http://some.example.com/downloads/foo.py#egg=foo-1.0" foo
```

will create a `.egg` file in the current directory.

## Creating your own Package Index

In addition to local directories and the Python Package Index, EasyInstall can find download links on most any web page whose URL is given to the `-f` (`--find-links`) option. In the simplest case, you can simply have a web page with links to eggs or Python source packages, even an automatically generated directory listing (such as the Apache web server provides).

If you are setting up an intranet site for package downloads, you may want to configure the target machines to use your download site by default, adding something like this to their *configuration files*:

```
[easy_install]
find_links = http://mypackages.example.com/somedir/
             http://turbogears.org/download/
             http://peak.telecommunity.com/dist/
```

As you can see, you can list multiple URLs separated by whitespace, continuing on multiple lines if necessary (as long as the subsequent lines are indented).

If you are more ambitious, you can also create an entirely custom package index or PyPI mirror. See the `--index-url` option under *Command-Line Options*, below, and also the section on *Package Index "API"*.

### 6.1.9 Password-Protected Sites

If a site you want to download from is password-protected using HTTP "Basic" authentication, you can specify your credentials in the URL, like so:

```
http://some_userid:some_password@some.example.com/some_path/
```

You can do this with both index page URLs and direct download URLs. As long as any HTML pages read by `easy_install` use *relative* links to point to the downloads, the same user ID and password will be used to do the downloading.



## 6.1.10 Using .pypirc Credentials

In addition to supplying credentials in the URL, `easy_install` will also honor credentials if present in the `.pypirc` file. Teams maintaining a private repository of packages may already have defined access credentials for uploading packages according to the `distutils` documentation. `easy_install` will attempt to honor those if present. Refer to the `distutils` documentation for Python 2.5 or later for details on the syntax.

## Controlling Build Options

EasyInstall respects standard `distutils` *Configuration Files*, so you can use them to configure build options for packages that it installs from source. For example, if you are on Windows using the MinGW compiler, you can configure the default compiler by putting something like this:

```
[build]
compiler = mingw32
```

into the appropriate `distutils` configuration file. In fact, since this is just normal `distutils` configuration, it will affect any builds using that config file, not just ones done by EasyInstall. For example, if you add those lines to `distutils.cfg` in the `distutils` package directory, it will be the default compiler for *all* packages you build. See *Configuration Files* below for a list of the standard configuration file locations, and links to more documentation on using `distutils` configuration files.

## Editing and Viewing Source Packages

Sometimes a package's source distribution contains additional documentation, examples, configuration files, etc., that are not part of its actual code. If you want to be able to examine these files, you can use the `--editable` option to EasyInstall, and EasyInstall will look for a source distribution or Subversion URL for the package, then download and extract it or check it out as a subdirectory of the `--build-directory` you specify. If you then wish to install the package after editing or configuring it, you can do so by rerunning EasyInstall with that directory as the target.

Note that using `--editable` stops EasyInstall from actually building or installing the package; it just finds, obtains, and possibly unpacks it for you. This allows you to make changes to the package if necessary, and to either install it in development mode using `setup.py develop` (if the package uses `setuptools`, that is), or by running `easy_install projectdir` (where `projectdir` is the subdirectory EasyInstall created for the downloaded package).

In order to use `--editable` (`-e` for short), you *must* also supply a `--build-directory` (`-b` for short). The project will be placed in a subdirectory of the build directory. The subdirectory will have the same name as the project itself, but in all-lowercase. If a file or directory of that name already exists, EasyInstall will print an error message and exit.

Also, when using `--editable`, you cannot use URLs or filenames as arguments. You *must* specify project names (and optional version requirements) so that EasyInstall knows what directory name(s) to create. If you need to force EasyInstall to use a particular URL or filename, you should specify it as a `--find-links` item (`-f` for short), and then also specify the project name, e.g.:

```
easy_install -eb ~/projects \
-fhttp://prdownloads.sourceforge.net/ctypes/ctypes-0.9.6.tar.gz?download \
ctypes==0.9.6
```

## Dealing with Installation Conflicts

(NOTE: As of 0.6a11, this section is obsolete; it is retained here only so that people using older versions of EasyInstall can consult it. As of version 0.6a11, installation conflicts are handled automatically without deleting the old or system-

installed packages, and without ignoring the issue. Instead, eggs are automatically shifted to the front of `sys.path` using special code added to the `easy-install.pth` file. So, if you are using version 0.6a11 or better of setuptools, you do not need to worry about conflicts, and the following issues do not apply to you.)

EasyInstall installs distributions in a “managed” way, such that each distribution can be independently activated or deactivated on `sys.path`. However, packages that were not installed by EasyInstall are “unmanaged”, in that they usually live all in one directory and cannot be independently activated or deactivated.

As a result, if you are using EasyInstall to upgrade an existing package, or to install a package with the same name as an existing package, EasyInstall will warn you of the conflict. (This is an improvement over `setup.py install`, because the `distutils` just install new packages on top of old ones, possibly combining two unrelated packages or leaving behind modules that have been deleted in the newer version of the package.)

EasyInstall will stop the installation if it detects a conflict between an existing, “unmanaged” package, and a module or package in any of the distributions you’re installing. It will display a list of all of the existing files and directories that would need to be deleted for the new package to be able to function correctly. To proceed, you must manually delete these conflicting files and directories and re-run EasyInstall.

Of course, once you’ve replaced all of your existing “unmanaged” packages with versions managed by EasyInstall, you won’t have any more conflicts to worry about!

## Compressed Installation

EasyInstall tries to install packages in zipped form, if it can. Zipping packages can improve Python’s overall import performance if you’re not using the `--multi-version` option, because Python processes zipfile entries on `sys.path` much faster than it does directories.

As of version 0.5a9, EasyInstall analyzes packages to determine whether they can be safely installed as a zipfile, and then acts on its analysis. (Previous versions would not install a package as a zipfile unless you used the `--zip-ok` option.)

The current analysis approach is fairly conservative; it currently looks for:

- Any use of the `__file__` or `__path__` variables (which should be replaced with `pkg_resources` API calls)
- Possible use of `inspect` functions that expect to manipulate source files (e.g. `inspect.getsource()`)
- Top-level modules that might be scripts used with `python -m` (Python 2.4)

If any of the above are found in the package being installed, EasyInstall will assume that the package cannot be safely run from a zipfile, and unzip it to a directory instead. You can override this analysis with the `-zip-ok` flag, which will tell EasyInstall to install the package as a zipfile anyway. Or, you can use the `--always-unzip` flag, in which case EasyInstall will always unzip, even if its analysis says the package is safe to run as a zipfile.

Normally, however, it is simplest to let EasyInstall handle the determination of whether to zip or unzip, and only specify overrides when needed to work around a problem. If you find you need to override EasyInstall’s guesses, you may want to contact the package author and the EasyInstall maintainers, so that they can make appropriate changes in future versions.

(Note: If a package uses `setuptools` in its setup script, the package author has the option to declare the package safe or unsafe for zipped usage via the `zip_safe` argument to `setup()`. If the package author makes such a declaration, EasyInstall believes the package’s author and does not perform its own analysis. However, your command-line option, if any, will still override the package author’s choice.)

## 6.2 Reference Manual

### 6.2.1 Configuration Files

(New in 0.4a2)

You may specify default options for EasyInstall using the standard distutils configuration files, under the command heading `easy_install`. EasyInstall will look first for a `setup.cfg` file in the current directory, then a `~/.pydistutils.cfg` or `$HOME\pydistutils.cfg` (on Unix-like OSes and Windows, respectively), and finally a `distutils.cfg` file in the `distutils` package directory. Here's a simple example:

```
[easy_install]

# set the default location to install packages
install_dir = /home/me/lib/python

# Notice that indentation can be used to continue an option
# value; this is especially useful for the "--find-links"
# option, which tells easy_install to use download links on
# these pages before consulting PyPI:
#
find_links = http://sqlobject.org/
             http://peak.telecommunity.com/dist/
```

In addition to accepting configuration for its own options under `[easy_install]`, EasyInstall also respects defaults specified for other distutils commands. For example, if you don't set an `install_dir` for `[easy_install]`, but *have* set an `install_lib` for the `[install]` command, this will become EasyInstall's default installation directory. Thus, if you are already using distutils configuration files to set default install locations, build options, etc., EasyInstall will respect your existing settings until and unless you override them explicitly in an `[easy_install]` section.

For more information, see also the current Python documentation on the [use and location of distutils configuration files](#).

Notice that `easy_install` will use the `setup.cfg` from the current working directory only if it was triggered from `setup.py` through the `install_requires` option. The standalone command will not use that file.

### 6.2.2 Command-Line Options

**--zip-ok, -z** Install all packages as zip files, even if they are marked as unsafe for running as a zipfile. This can be useful when EasyInstall's analysis of a non-setuptools package is too conservative, but keep in mind that the package may not work correctly. (Changed in 0.5a9; previously this option was required in order for zipped installation to happen at all.)

**--always-unzip, -Z** Don't install any packages as zip files, even if the packages are marked as safe for running as a zipfile. This can be useful if a package does something unsafe, but not in a way that EasyInstall can easily detect. EasyInstall's default analysis is currently very conservative, however, so you should only use this option if you've had problems with a particular package, and *after* reporting the problem to the package's maintainer and to the EasyInstall maintainers.

(Note: the `-z/-Z` options only affect the installation of newly-built or downloaded packages that are not already installed in the target directory; if you want to convert an existing installed version from zipped to unzipped or vice versa, you'll need to delete the existing version first, and re-run EasyInstall.)

**--multi-version, -m** "Multi-version" mode. Specifying this option prevents `easy_install` from adding an `easy-install.pth` entry for the package being installed, and if an entry for any version the package

already exists, it will be removed upon successful installation. In multi-version mode, no specific version of the package is available for importing, unless you use `pkg_resources.require()` to put it on `sys.path`. This can be as simple as:

```
from pkg_resources import require
require("SomePackage", "OtherPackage", "MyPackage")
```

which will put the latest installed version of the specified packages on `sys.path` for you. (For more advanced uses, like selecting specific versions and enabling optional dependencies, see the `pkg_resources` API doc.)

Changed in 0.6a10: this option is no longer silently enabled when installing to a non-PYTHONPATH, non-“site” directory. You must always explicitly use this option if you want it to be active.

**--upgrade, -U (New in 0.5a4)** By default, EasyInstall only searches online if a project/version requirement can't be met by distributions already installed on `sys.path` or the installation directory. However, if you supply the `--upgrade` or `-U` flag, EasyInstall will always check the package index and `--find-links` URLs before selecting a version to install. In this way, you can force EasyInstall to use the latest available version of any package it installs (subject to any version requirements that might exclude such later versions).

**--install-dir=DIR, -d DIR** Set the installation directory. It is up to you to ensure that this directory is on `sys.path` at runtime, and to use `pkg_resources.require()` to enable the installed package(s) that you need.

(New in 0.4a2) If this option is not directly specified on the command line or in a `distutils` configuration file, the `distutils` default installation location is used. Normally, this would be the `site-packages` directory, but if you are using `distutils` configuration files, setting things like `prefix` or `install_lib`, then those settings are taken into account when computing the default installation directory, as is the `--prefix` option.

**--script-dir=DIR, -s DIR** Set the script installation directory. If you don't supply this option (via the command line or a configuration file), but you *have* supplied an `--install-dir` (via command line or config file), then this option defaults to the same directory, so that the scripts will be able to find their associated package installation. Otherwise, this setting defaults to the location where the `distutils` would normally install scripts, taking any `distutils` configuration file settings into account.

**--exclude-scripts, -x** Don't install scripts. This is useful if you need to install multiple versions of a package, but do not want to reset the version that will be run by scripts that are already installed.

**--user (New in 0.6.11)** Use the user-site-packages as specified in [PEP 370](#) instead of the global site-packages.

**--always-copy, -a (New in 0.5a4)** Copy all needed distributions to the installation directory, even if they are already present in a directory on `sys.path`. In older versions of EasyInstall, this was the default behavior, but now you must explicitly request it. By default, EasyInstall will no longer copy such distributions from other `sys.path` directories to the installation directory, unless you explicitly gave the distribution's filename on the command line.

Note that as of 0.6a10, using this option excludes “system” and “development” eggs from consideration because they can't be reliably copied. This may cause EasyInstall to choose an older version of a package than what you expected, or it may cause downloading and installation of a fresh copy of something that's already installed. You will see warning messages for any eggs that EasyInstall skips, before it falls back to an older version or attempts to download a fresh copy.

**--find-links=URLS\_OR\_FILENAMES, -f URLS\_OR\_FILENAMES** Scan the specified “download pages” or directories for direct links to eggs or other distributions. Any existing file or directory names or direct download URLs are immediately added to EasyInstall's search cache, and any indirect URLs (ones that don't point to eggs or other recognized archive formats) are added to a list of additional places to search for download links. As soon as EasyInstall has to go online to find a package (either because it doesn't exist locally, or because `--upgrade` or `-U` was used), the specified URLs will be downloaded and scanned for additional direct links.

Eggs and archives found by way of `--find-links` are only downloaded if they are needed to meet a requirement specified on the command line; links to unneeded packages are ignored.

If all requested packages can be found using links on the specified download pages, the Python Package Index will not be consulted unless you also specified the `--upgrade` or `-U` option.

(Note: if you want to refer to a local HTML file containing links, you must use a `file: URL`, as filenames that do not refer to a directory, egg, or archive are ignored.)

You may specify multiple URLs or file/directory names with this option, separated by whitespace. Note that on the command line, you will probably have to surround the URL list with quotes, so that it is recognized as a single option value. You can also specify URLs in a configuration file; see *Configuration Files*, above.

Changed in 0.6a10: previously all URLs and directories passed to this option were scanned as early as possible, but from 0.6a10 on, only directories and direct archive links are scanned immediately; URLs are not retrieved unless a package search was already going to go online due to a package not being available locally, or due to the use of the `--update` or `-U` option.

**--no-find-links** **Blocks the addition of any link.** This parameter is useful if you want to avoid adding links defined in a project `easy_install` is installing (whether it's a requested project or a dependency). When used, `--find-links` is ignored.

Added in Distribute 0.6.11 and Setuptools 0.7.

**--index-url=URL, -i URL (New in 0.4a1; default changed in 0.6c7)** Specifies the base URL of the Python Package Index. The default is <https://pypi.org/simple/> if not specified. When a package is requested that is not locally available or linked from a `--find-links` download page, the package index will be searched for download pages for the needed package, and those download pages will be searched for links to download an egg or source distribution.

**--editable, -e (New in 0.6a1)** Only find and download source distributions for the specified projects, unpacking them to subdirectories of the specified `--build-directory`. EasyInstall will not actually build or install the requested projects or their dependencies; it will just find and extract them for you. See *Editing and Viewing Source Packages* above for more details.

**--build-directory=DIR, -b DIR (UPDATED in 0.6a1)** Set the directory used to build source packages. If a package is built from a source distribution or checkout, it will be extracted to a subdirectory of the specified directory. The subdirectory will have the same name as the extracted distribution's project, but in all-lowercase. If a file or directory of that name already exists in the given directory, a warning will be printed to the console, and the build will take place in a temporary directory instead.

This option is most useful in combination with the `--editable` option, which forces EasyInstall to *only* find and extract (but not build and install) source distributions. See *Editing and Viewing Source Packages*, above, for more information.

**--verbose, -v, --quiet, -q (New in 0.4a4)** Control the level of detail of EasyInstall's progress messages. The default detail level is "info", which prints information only about relatively time-consuming operations like running a setup script, unpacking an archive, or retrieving a URL. Using `-q` or `--quiet` drops the detail level to "warn", which will only display installation reports, warnings, and errors. Using `-v` or `--verbose` increases the detail level to include individual file-level operations, link analysis messages, and distutils messages from any setup scripts that get run. If you include the `-v` option more than once, the second and subsequent uses are passed down to any setup scripts, increasing the verbosity of their reporting as well.

**--dry-run, -n (New in 0.4a4)** Don't actually install the package or scripts. This option is passed down to any setup scripts run, so packages should not actually build either. This does *not* skip downloading, nor does it skip extracting source distributions to a temporary/build directory.

**--optimize=LEVEL, -O LEVEL (New in 0.4a4)** If you are installing from a source distribution, and are *not* using the `--zip-ok` option, this option controls the optimization level for compiling installed `.py` files to `.pyo` files. It does not affect the compilation of modules contained in `.egg` files, only those in `.egg` directories. The optimization level can be set to 0, 1, or 2; the default is 0 (unless it's set under `install` or `install_lib` in one of your distutils configuration files).

**--record=FILENAME (New in 0.5a4)** Write a record of all installed files to FILENAME. This is basically the same as the same option for the standard distutils “install” command, and is included for compatibility with tools that expect to pass this option to “setup.py install”.

**--site-dirs=DIRLIST, -S DIRLIST (New in 0.6a1)** Specify one or more custom “site” directories (separated by commas). “Site” directories are directories where .pth files are processed, such as the main Python site-packages directory. As of 0.6a10, EasyInstall automatically detects whether a given directory processes .pth files (or can be made to do so), so you should not normally need to use this option. It is now only necessary if you want to override EasyInstall’s judgment and force an installation directory to be treated as if it supported .pth files.

**--no-deps, -N (New in 0.6a6)** Don’t install any dependencies. This is intended as a convenience for tools that wrap eggs in a platform-specific packaging system. (We don’t recommend that you use it for anything else.)

**--allow-hosts=PATTERNS, -H PATTERNS (New in 0.6a6)** Restrict downloading and spidering to hosts matching the specified glob patterns. E.g. -H \*.python.org restricts web access so that only packages listed and downloadable from machines in the python.org domain. The glob patterns must match the *entire* user/host/port section of the target URL(s). For example, \*.python.org will NOT accept a URL like http://python.org/foo or http://www.python.org:8080/. Multiple patterns can be specified by separating them with commas. The default pattern is \*, which matches anything.

In general, this option is mainly useful for blocking EasyInstall’s web access altogether (e.g. -Hlocalhost), or to restrict it to an intranet or other trusted site. EasyInstall will do the best it can to satisfy dependencies given your host restrictions, but of course can fail if it can’t find suitable packages. EasyInstall displays all blocked URLs, so that you can adjust your --allow-hosts setting if it is more strict than you intended. Some sites may wish to define a restrictive default setting for this option in their *configuration files*, and then manually override the setting on the command line as needed.

**--prefix=DIR (New in 0.6a10)** Use the specified directory as a base for computing the default installation and script directories. On Windows, the resulting default directories will be prefix\\Lib\\site-packages and prefix\\Scripts, while on other platforms the defaults will be prefix/lib/python2.X/site-packages (with the appropriate version substituted) for libraries and prefix/bin for scripts.

Note that the --prefix option only sets the *default* installation and script directories, and does not override the ones set on the command line or in a configuration file.

**--local-snapshots-ok, -l (New in 0.6c6)** Normally, EasyInstall prefers to only install *released* versions of projects, not in-development ones, because such projects may not have a currently-valid version number. So, it usually only installs them when their setup.py directory is explicitly passed on the command line.

However, if this option is used, then any in-development projects that were installed using the setup.py develop command, will be used to build eggs, effectively upgrading the “in-development” project to a snapshot release. Normally, this option is used only in conjunction with the --always-copy option to create a distributable snapshot of every egg needed to run an application.

Note that if you use this option, you must make sure that there is a valid version number (such as an SVN revision number tag) for any in-development projects that may be used, as otherwise EasyInstall may not be able to tell what version of the project is “newer” when future installations or upgrades are attempted.

## 6.2.3 Custom Installation Locations

By default, EasyInstall installs python packages into Python’s main site-packages directory, and manages them using a custom .pth file in that same directory.

Very often though, a user or developer wants easy\_install to install and manage python packages in an alternative location, usually for one of 3 reasons:

1. They don’t have access to write to the main Python site-packages directory.



2. They want a user-specific stash of packages, that is not visible to other users.
3. They want to isolate a set of packages to a specific python application, usually to minimize the possibility of version conflicts.

Historically, there have been many approaches to achieve custom installation. The following section lists only the easiest and most relevant approaches<sup>1</sup>.

*Use the “`-user`” option*

*Use the “`-user`” option and customize “`PYTHONUSERBASE`”*

*Use “`virtualenv`”*

## Use the “`-user`” option

Python provides a User scheme for installation, which means that all python distributions support an alternative install location that is specific to a user<sup>3</sup>. The Default location for each OS is explained in the python documentation for the `site.USER_BASE` variable. This mode of installation can be turned on by specifying the `--user` option to `setup.py install` or `easy_install`. This approach serves the need to have a user-specific stash of packages.

## Use the “`-user`” option and customize “`PYTHONUSERBASE`”

The User scheme install location can be customized by setting the `PYTHONUSERBASE` environment variable, which updates the value of `site.USER_BASE`. To isolate packages to a specific application, simply set the OS environment of that application to a specific value of `PYTHONUSERBASE`, that contains just those packages.

## Use “`virtualenv`”

“`virtualenv`” is a 3rd-party python package that effectively “clones” a python installation, thereby creating an isolated location to install packages. The evolution of “`virtualenv`” started before the existence of the User installation scheme. “`virtualenv`” provides a version of `easy_install` that is scoped to the cloned python install and is used in the normal way. “`virtualenv`” does offer various features that the User installation scheme alone does not provide, e.g. the ability to hide the main python site-packages.

Please refer to the [virtualenv](#) documentation for more details.

## 6.2.4 Package Index “API”

Custom package indexes (and PyPI) must follow the following rules for EasyInstall to be able to look up and download packages:

1. Except where stated otherwise, “pages” are HTML or XHTML, and “links” refer to `href` attributes.
2. Individual project version pages’ URLs must be of the form `base/projectname/version`, where `base` is the package index’s base URL.
3. Omitting the `/version` part of a project page’s URL (but keeping the trailing `/`) should result in a page that is either:
  - a) The single active version of that project, as though the version had been explicitly included, OR
  - b) A page with links to all of the active version pages for that project.

<sup>1</sup> There are older ways to achieve custom installation using various `easy_install` and `setup.py install` options, combined with `PYTHONPATH` and/or `PYTHONUSERBASE` alterations, but all of these are effectively deprecated by the User scheme brought in by PEP-370.

<sup>3</sup> Prior to the User scheme, there was the Home scheme, which is still available, but requires more effort than the User scheme to get packages recognized.

4. Individual project version pages should contain direct links to downloadable distributions where possible. It is explicitly permitted for a project's "long\_description" to include URLs, and these should be formatted as HTML links by the package index, as EasyInstall does no special processing to identify what parts of a page are index-specific and which are part of the project's supplied description.
5. Where available, MD5 information should be added to download URLs by appending a fragment identifier of the form `#md5=...`, where `...` is the 32-character hex MD5 digest. EasyInstall will verify that the downloaded file's MD5 digest matches the given value.
6. Individual project version pages should identify any "homepage" or "download" URLs using `rel="homepage"` and `rel="download"` attributes on the HTML elements linking to those URLs. Use of these attributes will cause EasyInstall to always follow the provided links, unless it can be determined by inspection that they are downloadable distributions. If the links are not to downloadable distributions, they are retrieved, and if they are HTML, they are scanned for download links. They are *not* scanned for additional "homepage" or "download" links, as these are only processed for pages that are part of a package index site.
7. The root URL of the index, if retrieved with a trailing `/`, must result in a page containing links to *all* projects' active version pages.  
  
(Note: This requirement is a workaround for the absence of case-insensitive `safe_name()` matching of project names in URL paths. If project names are matched in this fashion (e.g. via the PyPI server, `mod_rewrite`, or a similar mechanism), then it is not necessary to include this all-packages listing page.)
8. If a package index is accessed via a `file://` URL, then EasyInstall will automatically use `index.html` files, if present, when trying to read a directory with a trailing `/` on the URL.



### 7.1 v46.1.3

25 Mar 2020

No significant changes.

### 7.2 v46.1.2

25 Mar 2020

- [#1458](#): Added template for reporting Python 2 incompatibilities.

### 7.3 v46.1.1

21 Mar 2020

No significant changes.

### 7.4 v46.1.0

21 Mar 2020

- [#308](#): Allow version number normalization to be bypassed by wrapping in a `'setuptools.sic()'` call.
- [#1424](#): Prevent keeping files mode for `package_data` build. It may break a build if user's package data has read only flag.
- [#1431](#): In `easy_install.check_site_dir`, ensure the installation directory exists.
- [#1563](#): In `pkg_resources` prefer `find_spec` ([PEP 451](#)) to `find_module`.

Incorporate changes from v44.1.0:

- [#1704](#): Set `sys.argv[0]` in setup script run by `build_meta.__legacy__`
- [#1959](#): Fix for Python 4: replace unsafe `six.PY3` with `six.PY2`
- [#1994](#): Fixed a bug in the “`setuptools.finalize_distribution_options`” hook that lead to ignoring the order attribute of entry points managed by this hook.

## 7.5 v44.1.0

21 Mar 2020

- [#1704](#): Set `sys.argv[0]` in setup script run by `build_meta.__legacy__`
- [#1959](#): Fix for Python 4: replace unsafe `six.PY3` with `six.PY2`
- [#1994](#): Fixed a bug in the “`setuptools.finalize_distribution_options`” hook that lead to ignoring the order attribute of entry points managed by this hook.

## 7.6 v46.0.0

08 Mar 2020

- [#65](#): Once again as in 3.0, removed the Features feature.
- [#1890](#): Fix vendored dependencies so importing `setuptools.extern.some_module` gives the same object as `setuptools._vendor.some_module`. This makes Metadata picklable again.
- [#1899](#): Test suite now fails on warnings.
- [#2011](#): Fix broken link to distutils docs on `package_data`
- [#1991](#): Include `pkg_resources` test data in `sdist`, so tests can be executed from it.

## 7.7 v45.3.0

07 Mar 2020

- [#1557](#): Deprecated `eggsecutable` scripts and updated docs.
- [#1904](#): Update `msvc.py` to use CPython 3.8.0 mechanism to find `msvc` 14+

## 7.8 v45.2.0

08 Feb 2020

- [#1905](#): Fixed defect in `_imp`, introduced in 41.6.0 when the ‘tests’ directory is not present.
- [#1941](#): Improve editable installs with [PEP 518](#) build isolation:
  - The `--user` option is now always available. A warning is issued if the user site directory is not available.
  - The error shown when the install directory is not in `PYTHONPATH` has been turned into a warning.
- [#1981](#): Setuptools now declares its `tests` and `docs` dependencies in metadata (extras).

- [#1985](#): Add support for installing scripts in environments where `bdist_wininst` is missing (i.e. Python 3.9).
- [#1968](#): Add `flake8-2020` to check for misuse of `sys.version` or `sys.version_info`.

## 7.9 v45.1.0

19 Jan 2020

- [#1458](#): Add minimum sunset date and preamble to Python 2 warning.
- [#1704](#): Set `sys.argv[0]` in setup script run by `build_meta.__legacy__`
- [#1974](#): Add Python 3 Only Trove Classifier and remove universal wheel declaration for more complete transition from Python 2.

## 7.10 v45.0.0

11 Jan 2020

- [#1458](#): Drop support for Python 2. Setuptools now requires Python 3.5 or later. Install setuptools using `pip >=9` or `pin` to Setuptools <45 to maintain 2.7 support.
- [#1959](#): Fix for Python 4: replace unsafe `six.PY3` with `six.PY2`

## 7.11 v44.0.0

01 Jan 2020

- [#1908](#): Drop support for Python 3.4.

## 7.12 v43.0.0

31 Dec 2019

- [#1634](#): Include `pyproject.toml` in source distribution by default. Projects relying on the previous behavior where `pyproject.toml` was excluded by default should stop relying on that behavior or add `exclude pyproject.toml` to their `MANIFEST.in` file.
- [#1927](#): Setuptools once again declares ‘setuptools’ in the `build-system.requires` and adds [PEP 517](#) build support by declaring itself as the `build-backend`. It additionally specifies `build-system.backend-path` to rely on itself for those builders that support it.

## 7.13 v42.0.2

01 Dec 2019

- [#1921](#): Fix support for `easy_install`’s `find-links` option in `setup.cfg`.
- [#1922](#): Build dependencies (`setup_requires` and `tests_require`) now install transitive dependencies indicated by `extras`.

## 7.14 v42.0.1

25 Nov 2019

- [#1918](#): Fix regression in handling wheels compatibility tags.

## 7.15 v42.0.0

23 Nov 2019

- [#1830](#), [#1909](#): Mark the `easy_install` script and `setuptools` command as deprecated, and use `pip` when available to fetch/build packages.
  - support for `python_requires`
  - better support for wheels (proper handling of priority with respect to [PEP 425](#) tags)
  - [PEP 517/518](#) support
  - eggs are not supported
  - no support for the `allow_hosts` `easy_install` option (`index_url/find_links` are still honored)
  - `pip` environment variables are honored (and take precedence over `easy_install` options)
- [#1898](#): Removed the “upload” and “register” commands in favor of `twine`.
- [#1767](#): Add support for the `license_files` option in `setup.cfg` to automatically include multiple license files in a source distribution.
- [#1829](#): Update handling of wheels compatibility tags: \* add support for `manylinux2010` \* fix use of removed ‘m’ ABI flag in Python 3.8 on Windows
- [#1861](#): Fix empty namespace package installation from wheel.
- [#1877](#): Setuptools now exposes a new entry point hook “`setuptools.finalize_distribution_options`”, enabling plugins like `setuptools_scm` to configure options on the distribution at finalization time.

## 7.16 v41.6.0

29 Oct 2019

- [#479](#): Replace usage of deprecated `imp` module with local re-implementation in `setuptools._imp`.

## 7.17 v41.5.1

28 Oct 2019

- [#1891](#): Fix code for detecting Visual Studio’s version on Windows under Python 2.

## 7.18 v41.5.0

27 Oct 2019

- [#1811](#): Improve Visual C++ 14.X support, mainly for Visual Studio 2017 and 2019.
- [#1814](#): Fix `pkg_resources.Requirement` hash/equality implementation: take [PEP 508](#) direct URL into account.
- [#1824](#): Fix tests when running under `python3.10`.
- [#1878](#): Formally deprecated the `test` command, with the recommendation that users migrate to `tox`.
- [#1860](#): Update documentation to mention the egg format is not supported by pip and dependency links support was dropped starting with pip 19.0.
- [#1862](#): Drop `ez_setup` documentation: deprecated for some time (last updated in 2016), and still relying on `easy_install` (deprecated too).
- [#1868](#): Drop most documentation references to (deprecated) EasyInstall.
- [#1884](#): Added a trove classifier to document support for Python 3.8.
- [#1886](#): Added Python 3.8 release to the Travis test matrix.

## 7.19 v41.4.0

06 Oct 2019

- [#1847](#): In declarative config, now traps errors when invalid `python_requires` values are supplied.

## 7.20 v41.3.0

06 Oct 2019

- [#1690](#): When storing extras, rely on `OrderedSet` to retain order of extras as indicated by the packager, which will also be deterministic on Python 2.7 (with `PYTHONHASHSEED` unset) and Python 3.6+.
- [#1858](#): Fixed failing integration test triggered by `'long_description_content_type'` in packaging.

## 7.21 v41.2.0

21 Aug 2019

- [#479](#): Remove some usage of the deprecated `imp` module.
- [#1565](#): Changed `html_sidebars` from string to list of string as per <https://www.sphinx-doc.org/en/master/changes.html#id58>

## 7.22 v41.1.0

13 Aug 2019

- [#1697](#): Moved most of the constants from `setup.py` to `setup.cfg`

- [#1749](#): Fixed issue with the [PEP 517](#) backend where building a source distribution would fail if any tarball existed in the destination directory.
- [#1750](#): Fixed an issue with [PEP 517](#) backend where wheel builds would fail if the destination directory did not already exist.
- [#1756](#): Force metadata-version  $\geq$  1.2. when project urls are present.
- [#1769](#): Improve `package_data` check: ensure the dictionary values are lists/tuples of strings.
- [#1788](#): Changed compatibility fallback logic for `html.unescape` to avoid accessing `HTMLParser.unescape` when not necessary. `HTMLParser.unescape` is deprecated and will be removed in Python 3.9.
- [#1790](#): Added the file path to the error message when a `UnicodeDecodeError` occurs while reading a metadata file.
- [#1776](#): Use license classifiers rather than the license field.

## 7.23 v41.0.1

22 Apr 2019

- [#1671](#): Fixed issue with the [PEP 517](#) backend that prevented building a wheel when the `dist/` directory contained existing `.whl` files.
- [#1709](#): In `test.paths_on_python_path`, avoid adding unnecessary duplicates to the `PYTHONPATH`.
- [#1741](#): In `package_index`, now honor “current directory” during a checkout of git and hg repositories under Windows

## 7.24 v41.0.0

05 Apr 2019

- [#1735](#): When parsing `setup.cfg` files, setuptools now requires the files to be encoded as UTF-8. Any other encoding will lead to a `UnicodeDecodeError`. This change removes support for specifying an encoding using a `'coding: '` directive in the header of the file, a feature that was introduced in 40.7. Given the recent release of the aforementioned feature, it is assumed that few if any projects are utilizing the feature to specify an encoding other than UTF-8.

## 7.25 v40.9.0

03 Apr 2019

- [#1675](#): Added support for `setup.cfg`-only projects when using the `setuptools.build_meta` backend. Projects that have enabled [PEP 517](#) no longer need to have a `setup.py` and can use the purely declarative `setup.cfg` configuration file instead.
- [#1720](#): Added support for `pkg_resources.parse_requirements`-style requirements in `setup_requires` when `setup.py` is invoked from the `setuptools.build_meta` build backend.
- [#1664](#): Added the path to the `PKG-INFO` or `METADATA` file in the exception text when the `Version:` header can't be found.

- [#1705](#): Removed some placeholder documentation sections referring to deprecated features.

## 7.26 v40.8.0

05 Feb 2019

- [#1652](#): Added the `build_meta: __legacy__` backend, a “compatibility mode” [PEP 517](#) backend that can be used as the default when `build-backend` is left unspecified in `pyproject.toml`.
- [#1635](#): Resource paths are passed to `pkg_resources.resource_string` and similar no longer accept paths that traverse parents, that begin with a leading `/`. Violations of this expectation raise `DeprecationWarnings` and will become errors. Additionally, any paths that are absolute on Windows are strictly disallowed and will raise `ValueErrors`.
- [#1536](#): `setuptools` will now automatically include licenses if `setup.cfg` contains a `license_file` attribute, unless this file is manually excluded inside `MANIFEST.in`.

## 7.27 v40.7.3

03 Feb 2019

- [#1670](#): In `package_index`, revert to using a copy of `splituser` from Python 3.8. Attempts to use `urllib.parse.urlparse` led to problems as reported in [#1663](#) and [#1668](#). This change serves as an alternative to [#1499](#) and fixes [#1668](#).

## 7.28 v40.7.2

31 Jan 2019

- [#1666](#): Restore port in URL handling in `package_index`.

## 7.29 v40.7.1

28 Jan 2019

- [#1660](#): On Python 2, when reading config files, downcast options from text to bytes to satisfy `distutils` expectations.

## 7.30 v40.7.0

27 Jan 2019

- [#1551](#): File inputs for the `license` field in `setup.cfg` files now explicitly raise an error.
- [#1180](#): Add support for non-ASCII in `setup.cfg` ([#1062](#)). Add support for native strings on some parameters ([#1136](#)).
- [#1499](#): `setuptools.package_index` no longer relies on the deprecated `urllib.parse.splituser` per [Python #27485](#).
- [#1544](#): Added tests for `PackageIndex.download` (for git URLs).

- [#1625](#): In [PEP 517](#) `build_meta` builder, ensure that sdist is built as gztar per the spec.

## 7.31 v40.6.3

11 Dec 2018

- [#1594](#): [PEP 517](#) backend no longer declares setuptools as a dependency as it can be assumed.

## 7.32 v40.6.2

13 Nov 2018

- [#1592](#): Fix invalid dependency on external six module (instead of vendored version).

## 7.33 v40.6.1

12 Nov 2018

- [#1590](#): Fixed regression where packages without `author` or `author_email` fields generated malformed package metadata.

## 7.34 v40.6.0

12 Nov 2018

- [#1541](#): Officially deprecated the `requires` parameter in `setup()`.
- [#1519](#): In `pkg_resources.normalize_path`, additional path normalization is now performed to ensure path values to a directory is always the same, preventing false positives when checking scripts have a consistent prefix to set up on Windows.
- [#1545](#): Changed the warning class of all deprecation warnings; deprecation warning classes are no longer derived from `DeprecationWarning` and are thus visible by default.
- [#1554](#): `build_meta.build_sdist` now includes `setup.py` in source distributions by default.
- [#1576](#): Started monkey-patching `get_metadata_version` and `read_pkg_file` onto `distutils.DistributionMetadata` to retain the correct version on the `PKG-INFO` file in the (deprecated) `upload` command.
- [#1533](#): Restricted the `recursive-include` `setuptools/_vendor` to contain only `.py` and `.txt` files.
- [#1395](#): Changed Pyrex references to Cython in the documentation.
- [#1456](#): Documented that the `rpmbuild` packages is required for the `bdist_rpm` command.
- [#1537](#): Documented how to use `setup.cfg` for `src/` layouts
- [#1539](#): Added minimum version column in `setup.cfg` metadata table.
- [#1552](#): Fixed a minor typo in the python 2/3 compatibility documentation.
- [#1553](#): Updated installation instructions to point to `pip install` instead of `ez_setup.py`.
- [#1560](#): Updated `setuptools` distribution documentation to remove some outdated information.



- [#1564](#): Documented `setup.cfg` minimum version for `version` and `project_urls`.
- [#1572](#): Added the `concurrent.futures` backport `futures` to the Python 2.7 test suite requirements.

## 7.35 v40.5.0

26 Oct 2018

- [#1335](#): In `pkg_resources.normalize_path`, fix issue on Cygwin when `cwd` contains symlinks.
- [#1502](#): Deprecated support for downloads from Subversion in `package_index/easy_install`.
- [#1517](#): Dropped use of `six.u` in favor of `u`““ literals.
- [#1520](#): Added support for `data_files` in `setup.cfg`.
- [#1525](#): Fixed rendering of the deprecation warning in `easy_install` doc.

## 7.36 v40.4.3

23 Sep 2018

- [#1480](#): Bump vendored `pyparsing` in `pkg_resources` to 2.2.1.

## 7.37 v40.4.2

21 Sep 2018

- [#1497](#): Updated `gitignore` in repo.

## 7.38 v40.4.1

18 Sep 2018

- [#1480](#): Bump vendored `pyparsing` to 2.2.1.

## 7.39 v40.4.0

18 Sep 2018

- [#1481](#): Join the `sdist --dist-dir` and the `build_meta sdist` directory argument to point to the same target (meaning the build frontend no longer needs to clean manually the `dist` dir to avoid multiple `sdist` presence, and `setuptools` no longer needs to handle conflicts between the two).

## 7.40 v40.3.0

16 Sep 2018

- [#1402](#): Fixed a bug with namespace packages under Python 3.6 when one package in current directory hides another which is installed.
- [#1427](#): Set timestamp of `.egg-info` directory whenever `egg_info` command is run.
- [#1474](#): `build_meta.get_requires_for_build_sdist` now does not include the wheel package anymore.
- [#1486](#): Suppress warnings in `pkg_resources.handle_ns`.
- [#1479](#): Remove internal use of `six.binary_type`.

## 7.41 v40.2.0

21 Aug 2018

- [#1466](#): Fix handling of Unicode arguments in [PEP 517](#) backend

## 7.42 v40.1.1

21 Aug 2018

- [#1465](#): Fix regression with `egg_info` command when tagging is used.

## 7.43 v40.1.0

17 Aug 2018

- [#1410](#): Deprecated `upload` and `register` commands.
- [#1312](#): Introduced `find_namespace_packages()` to find [PEP 420](#) namespace packages.
- [#1420](#): Added `find_namespace:` directive to config parser.
- [#1418](#): Solved race in when creating egg cache directories.
- [#1450](#): Upgraded vendored PyParsing from 2.1.10 to 2.2.0.
- [#1451](#): Upgraded vendored appdirs from 1.4.0 to 1.4.3.
- [#1388](#): Fixed “Microsoft Visual C++ Build Tools” link in exception when Visual C++ not found.
- [#1389](#): Added support for scripts which have unicode content.
- [#1416](#): Moved several Python version checks over to using `six.PY2` and `six.PY3`.
- [#1441](#): Removed spurious executable permissions from files that don’t need them.

## 7.44 v40.0.0

09 Jul 2018

- [#1342](#): Drop support for Python 3.3.
- [#1366](#): In `package_index`, fixed handling of encoded entities in URLs.
- [#1383](#): In `pkg_resources VendorImporter`, avoid removing packages imported from the root.

- [#1379](#): Minor doc fixes after actually using the new release process.
- [#1385](#): Removed section on non-package data files.
- [#1403](#): Fix developer's guide.
- [#1404](#): Fix [PEP 518](#) configuration: set build requirements in `pyproject.toml` to `["wheel"]`.

## 7.45 v39.2.0

19 May 2018

- [#1359](#): Support using “file:” to load a [PEP 440](#)-compliant package version from a text file.
- [#1360](#): Fixed issue with a mismatch between the name of the package and the name of the `.dist-info` file in wheel files
- [#1364](#): Add `__dir__()` implementation to `pkg_resources.Distribution()` that includes the attributes in the `_provider` instance variable.
- [#1365](#): Take the `package_dir` option into account when loading the version from a module attribute.
- [#1353](#): Added coverage badge to README.
- [#1356](#): Made small fixes to the developer guide documentation.
- [#1357](#): Fixed warnings in documentation builds and started enforcing that the docs build without warnings in tox.
- [#1376](#): Updated release process docs.
- [#1343](#): The setuptools specific `long_description_content_type`, `project_urls` and `provides_extras` fields are now set consistently after any `distutils.setup_keywords` calls, allowing them to override values.
- [#1352](#): Added `tox` environment for documentation builds.
- [#1354](#): Added `towncrier` for changelog management.
- [#1355](#): Add PR template.
- [#1368](#): Fixed tests which failed without network connectivity.
- [#1369](#): Added unit tests for [PEP 425](#) compatibility tags support.
- [#1372](#): Stop testing Python 3.3 in Travis CI, now that the latest version of `wheel` no longer installs on it.

## 7.46 v39.1.0

28 Apr 2018

- [#1340](#): Update all PyPI URLs to reflect the switch to the new Warehouse codebase.
- [#1337](#): In `pkg_resources`, now support loading resources for modules loaded by the `SourcelessFileLoader`.
- [#1332](#): Silence spurious wheel related warnings on Windows.

## 7.47 v39.0.1

18 Mar 2018

- [#1297](#): Restore Unicode handling for Maintainer fields in metadata.

## 7.48 v39.0.0

17 Mar 2018

- [#1296](#): Setuptools now vendors its own direct dependencies, no longer relying on the dependencies as vendored by `pkg_resources`.
- [#296](#): Removed long-deprecated support for iteration on `Version` objects as returned by `pkg_resources.parse_version`. Removed the `SetuptoolsVersion` and `SetuptoolsLegacyVersion` names as well. They should not have been used, but if they were, replace with `Version` and `LegacyVersion` from `packaging.version`.

## 7.49 v38.7.0

17 Mar 2018

- [#1288](#): Add support for maintainer in PKG-INFO.

## 7.50 v38.6.1

17 Mar 2018

- [#1292](#): Avoid generating `Provides-Extra` in metadata when no extra is present (but environment markers are).

## 7.51 v38.6.0

15 Mar 2018

- [#1286](#): Add support for Metadata 2.1 ([PEP 566](#)).

## 7.52 v38.5.2

06 Mar 2018

- [#1285](#): Fixed `RuntimeError` in `pkg_resources.parse_requirements` on Python 3.7 (stemming from [PEP 479](#)).

## 7.53 v38.5.1

06 Feb 2018

- [#1271](#): Revert to Cython legacy `build_ext` behavior for compatibility.

## 7.54 v38.5.0

04 Feb 2018

- [#1229](#): Expand imports in `build_ext` to refine detection of Cython availability.
- [#1270](#): When Cython is available, `build_ext` now uses the new `build_ext`.

## 7.55 v38.4.1

03 Feb 2018

- [#1257](#): In `bdist_egg.scan_module`, fix `ValueError` on Python 3.7.

## 7.56 v38.4.0

05 Jan 2018

- [#1231](#): Removed warning when `PYTHONDONTWRITEBYTECODE` is enabled.

## 7.57 v38.3.0

04 Jan 2018

- [#1210](#): Add support for [PEP 345](#) Project-URL metadata.
- [#1207](#): Add support for `long_description_type` to `setup.cfg` declarative config as intended and documented.

## 7.58 v38.2.5

- [#1232](#): Fix trailing slash handling in `pkg_resources.ZipProvider`.

## 7.59 v38.2.4

04 Dec 2017

- [#1220](#): Fix `data_files` handling when installing from wheel.

## 7.60 v38.2.3

28 Nov 2017

- fix Travis' Python 3.3 job.

## 7.61 v38.2.2

27 Nov 2017

- [#1214](#): fix handling of namespace packages when installing from a wheel.

## 7.62 v38.2.1

26 Nov 2017

- [#1212](#): fix encoding handling of metadata when installing from a wheel.

## 7.63 v38.2.0

26 Nov 2017

- [#1200](#): easy\_install now support installing from wheels: they will be installed as standalone unzipped eggs.

## 7.64 v38.1.0

25 Nov 2017

- [#1208](#): Improve error message when failing to locate scripts in egg-info metadata.

## 7.65 v38.0.0

25 Nov 2017

- [#458](#): In order to support deterministic builds, Setuptools no longer allows packages to declare `install_requires` as unordered sequences (sets or dicts).

## 7.66 v37.0.0

20 Nov 2017

- [#878](#): Drop support for Python 2.6. Python 2.6 users should rely on 'setuptools < 37dev'.

## 7.67 v36.8.0

19 Nov 2017

- [#1190](#): In SSL support for package index operations, use SNI where available.

## 7.68 v36.7.3

13 Nov 2017

- [#1175](#): Bug fixes to `build_meta` module.

## 7.69 v36.7.2

13 Nov 2017

- [#701](#): Fixed duplicate test discovery on Python 3.

## 7.70 v36.7.1

11 Nov 2017

- [#1193](#): Avoid test failures in `bdist_egg` when `PYTHONDONTWRITEBYTECODE` is set.

## 7.71 v36.7.0

09 Nov 2017

- [#1054](#): Support `setup_requires` in `setup.cfg` files.

## 7.72 v36.6.1

09 Nov 2017

- [#1132](#): Removed redundant and costly serialization/parsing step in `EntryPoint.__init__`.
- [#844](#): `bdist_egg --exclude-source-files` now tested and works on Python 3.

## 7.73 v36.6.0

12 Oct 2017

- [#1143](#): Added `setuptools.build_meta` module, an implementation of [PEP-517](#) for Setuptools-defined packages.
- [#1143](#): Added `dist_info` command for producing `dist_info` metadata.

## 7.74 v36.5.0

15 Sep 2017

- [#170](#): When working with Mercurial checkouts, use Windows-friendly syntax for suppressing output.
- Inspired by [#1134](#), performed substantial refactoring of `pkg_resources.find_on_path` to facilitate an optimization for paths with many non-version entries.

## 7.75 v36.4.0

03 Sep 2017

- [#1075](#): Add new `Description-Content-Type` metadata field. [See here for documentation on how to use this field.](#)
- [#1068](#): Sort files and directories when building eggs for deterministic order.
- [#196](#): Remove caching of `easy_install` command in `fetch_build_egg`. Fixes issue where `pytest-runner-N.N` would satisfy the installation of `pytest`.
- [#1129](#): Fix working set dependencies handling when replacing conflicting distributions (e.g. when using `setup_requires` with a conflicting transitive dependency, fix [#1124](#)).
- [#1133](#): Improved handling of README files extensions and added Markdown to the list of searched READMEs.
- [#1135](#): Improve performance of `pkg_resources` import by not invoking `access` or `stat` and using `os.listdir` instead.

## 7.76 v36.3.0

28 Aug 2017

- [#1131](#): Make possible using several files within `file:` directive in `metadata.long_description` in `setup.cfg`.

## 7.77 v36.2.7

02 Aug 2017

- fix [#1105](#): Fix handling of requirements with environment markers when declared in `setup.cfg` (same treatment as for [#1081](#)).

## 7.78 v36.2.6

31 Jul 2017

- [#462](#): Don't assume a directory is an egg by the `.egg` extension alone.



## 7.79 v36.2.5

30 Jul 2017

- [#1093](#): Fix test command handler with `extras_require`.
- [#1112](#), [#1091](#), [#1115](#): Now using Trusty containers in Travis for CI and CD.

## 7.80 v36.2.4

26 Jul 2017

- [#1092](#): `pkg_resources` now uses `inspect.getmro` to resolve classes in method resolution order.

## 7.81 v36.2.3

25 Jul 2017

- [#1102](#): Restore behavior for empty extras.

## 7.82 v36.2.2

24 Jul 2017

- [#1099](#): Revert commit [a3ec721](#), restoring intended purpose of extras as part of a requirement declaration.

## 7.83 v36.2.1

23 Jul 2017

- fix [#1086](#)
- fix [#1087](#)
- support extras specifiers in `install_requires` requirements

## 7.84 v36.2.0

13 Jul 2017

- [#1081](#): Environment markers indicated in `install_requires` are now processed and treated as nameless `extras_require` with markers, allowing their metadata in `requires.txt` to be correctly generated.
- [#1053](#): Tagged commits are now released using Travis-CI build stages, meaning releases depend on passing tests on all supported Python versions (Linux) and not just the latest Python version.

## 7.85 v36.1.1

13 Jul 2017

- [#1083](#): Correct `py31compat.makedirs` to correctly honor `exist_ok` parameter.
- [#1083](#): Also use `makedirs` compatibility throughout `setuptools`.

## 7.86 v36.1.0

13 Jul 2017

- [#1083](#): Avoid race condition on directory creation in `pkg_resources.ensure_directory`.
- Removed deprecation of and restored support for `upload_docs` command for sites other than PyPI. Only warehouse is dropping support, but services like [devpi](#) continue to support docs built by `setuptools`' plugins. See [this comment](#) for more context on the motivation for this change.

## 7.87 v36.0.1

01 Jun 2017

- [#1042](#): Fix import in `py27compat` module that still referenced `six` directly, rather than through the `externs` module (vendored packages hook).

## 7.88 v36.0.0

30 May 2017

- [#980](#) and others: Once again, `Setuptools` vendors all of its dependencies. It seems to be the case that in the Python ecosystem, all build tools must run without any dependencies (build, runtime, or otherwise). At such a point that a mechanism exists that allows build tools to have dependencies, `Setuptools` will adopt it.

## 7.89 v35.0.2

27 Apr 2017

- [#1015](#): Fix test failures on Python 3.7.
- [#1024](#): Add workaround for [Jython #2581](#) in `monkey` module.

## 7.90 v35.0.1

18 Apr 2017

- [#992](#): Revert change introduced in `v34.4.1`, now considered invalid.
- [#1016](#): Revert change introduced in `v35.0.0` per [#1014](#), referencing [#436](#). The approach had unintended consequences, causing `sdist` installs to be missing files.

## 7.91 v35.0.0

15 Apr 2017

- [#436](#): In `egg_info.manifest_maker`, no longer read the file list from the manifest file, and instead re-build it on each build. In this way, files removed from the specification will not linger in the manifest. As a result, any files manually added to the manifest will be removed on subsequent `egg_info` invocations. No projects should be manually adding files to the manifest and should instead use `MANIFEST.in` or `SCM` file finders to force inclusion of files in the manifest.

## 7.92 v34.4.1

10 Apr 2017

- [#1008](#): In MSVC support, use always the last version available for Windows SDK and UCRT SDK.
- [#1008](#): In MSVC support, fix “`vcruntime140.dll`” returned path with Visual Studio 2017.
- [#992](#): In `msvc.msvc9_query_vcvarsall`, ensure the returned dicts have str values and not Unicode for compatibility with `os.environ`.

## 7.93 v34.4.0

07 Apr 2017

- [#995](#): In MSVC support, add support for “Microsoft Visual Studio 2017” and “Microsoft Visual Studio Build Tools 2017”.
- [#999](#) via [#1007](#): Extend support for declarative package config in a `setup.cfg` file to include the options `python_requires` and `py_modules`.

## 7.94 v34.3.3

26 Mar 2017

- [#967](#) (and [#997](#)): Explicitly import submodules of packaging to account for environments where the imports of those submodules is not implied by other behavior.

## 7.95 v34.3.2

11 Mar 2017

- [#993](#): Fix documentation upload by correcting rendering of content-type in `_build_multipart` on Python 3.

## 7.96 v34.3.1

02 Mar 2017

- [#988](#): Trap `os.unlink` same as `os.remove` in `auto_chmod` error handler.

- [#983](#): Fixes to invalid escape sequence deprecations on Python 3.6.

## 7.97 v34.3.0

23 Feb 2017

- [#941](#): In the upload command, if the username is blank, default to `getpass.getuser()`.
- [#971](#): Correct distutils findall monkeypatch to match appropriate versions (namely Python 3.4.6).

## 7.98 v34.2.0

12 Feb 2017

- [#966](#): Add support for reading dist-info metadata and thus locating Distributions from zip files.
- [#968](#): Allow '+' and '!' in egg fragments so that it can take package names that contain [PEP 440](#) conforming version specifiers.

## 7.99 v34.1.1

03 Feb 2017

- [#953](#): More aggressively employ the compatibility issue originally added in [#706](#).

## 7.100 v34.1.0

28 Jan 2017

- [#930](#): `build_info` now accepts two new parameters to optimize and customize the building of C libraries.

## 7.101 v34.0.3

28 Jan 2017

- [#947](#): Loosen restriction on the version of six required, restoring compatibility with environments relying on six 1.6.0 and later.

## 7.102 v34.0.2

24 Jan 2017

- [#882](#): Ensure extras are honored when building the working set.
- [#913](#): Fix issue in develop if package directory has a trailing slash.

## 7.103 v34.0.1

23 Jan 2017

- [#935](#): Fix glob syntax in graft.

## 7.104 v34.0.0

23 Jan 2017

- [#581](#): Instead of vendoring the growing list of dependencies that Setuptools requires to function, Setuptools now requires these dependencies just like any other project. Unlike other projects, however, Setuptools cannot rely on `setup_requires` to demand the dependencies it needs to install because its own machinery would be necessary to pull those dependencies if not present (a bootstrapping problem). As a result, Setuptools no longer supports self upgrade or installation in the general case. Instead, users are directed to use pip to install and upgrade using the `wheel` distributions of setuptools.

Users are welcome to contrive other means to install or upgrade Setuptools using other means, such as pre-installing the Setuptools dependencies with pip or a bespoke bootstrap tool, but such usage is not recommended and is not supported.

As discovered in [#940](#), not all versions of pip will successfully install Setuptools from its pre-built wheel. If you encounter issues with “No module named six” or “No module named packaging”, especially following a line “Running setup.py egg\_info for package setuptools”, then your pip is not new enough.

There’s an additional issue in pip where setuptools is upgraded concurrently with other source packages, described in pip [#4253](#). The proposed workaround is to always upgrade Setuptools first prior to upgrading other packages that would upgrade Setuptools.

## 7.105 v33.1.1

16 Jan 2017

- [#921](#): Correct issue where certifi fallback not being reached on Windows.

## 7.106 v33.1.0

15 Jan 2017

Installation via pip, as indicated in the [Python Packaging User’s Guide](#), is the officially-supported mechanism for installing Setuptools, and this recommendation is now explicit in the much more concise README.

Other edits and tweaks were made to the documentation. The codebase is unchanged.

## 7.107 v33.0.0

01 Jan 2017

- [#619](#): Removed support for the `tag_svn_revision` distribution option. If Subversion tagging support is still desired, consider adding the functionality to `setuptools_svn` in [setuptools\\_svn #2](#).

## 7.108 v32.3.1

28 Dec 2016

- [#866](#): Use `dis.Bytecode` on Python 3.4 and later in `setuptools.depends`.

## 7.109 v32.3.0

24 Dec 2016

- [#889](#): Backport proposed fix for disabling interpolation in `distutils.Distribution.parse_config_files`.

## 7.110 v32.2.0

22 Dec 2016

- [#884](#): Restore support for running the tests under `pytest-runner` by ensuring that `PYTHONPATH` is honored in tests invoking a subprocess.

## 7.111 v32.1.3

21 Dec 2016

- [#706](#): Add `rmtree` compatibility shim for environments where `rmtree` fails when passed a unicode string.

## 7.112 v32.1.2

18 Dec 2016

- [#893](#): Only release `sdist` in zip format as warehouse now disallows releasing two different formats.

## 7.113 v32.1.1

18 Dec 2016

- [#704](#): More selectively ensure that `'rmtree'` is not invoked with a byte string, enabling it to remove files that are non-ascii, even on Python 2.
- [#712](#): In `'sandbox.run_setup'`, ensure that `__file__` is always a `str`, modeling the behavior observed by the interpreter when invoking scripts and modules.

## 7.114 v32.1.0

16 Dec 2016

- [#891](#): In `'test'` command on test failure, raise `DistutilsError`, suppression invocation of subsequent commands.

## 7.115 v32.0.0

14 Dec 2016

- [#890](#): Revert [#849](#). `global-exclude .foo` will not match all `*.foo` files any more. Package authors must add an explicit wildcard, such as `global-exclude *.foo`, to match all `.foo` files. See [#886](#), [#849](#).

## 7.116 v31.0.1

14 Dec 2016

- [#885](#): Fix regression where `'pkg_resources._rebuild_mod_path'` would fail when a namespace package's `'__path__'` was not a list with a sort attribute.

## 7.117 v31.0.0

11 Dec 2016

- [#250](#): Install `'-nspkg.pth'` files for packages installed with `'setup.py develop'`. These `.pth` files allow namespace packages installed by pip or develop to co-mingle. This change required the removal of the change for [#805](#) and pip [#1924](#), introduced in 28.3.0 and implicated in [#870](#), but means that namespace packages not in a site packages directory will no longer work on Python earlier than 3.5, whereas before they would work on Python not earlier than 3.3.

## 7.118 v30.4.0

10 Dec 2016

- [#879](#): For declarative config:
  - `read_configuration()` now accepts `ignore_option_errors` argument. This allows scraping tools to read meta-data without a need to download entire packages. E.g. we can gather some stats right from GitHub repos just by downloading `setup.cfg`.
  - `packages find: directive` now supports fine tuning from a subsection. The same arguments as for `find()` are accepted.

## 7.119 v30.3.0

08 Dec 2016

- [#394](#) via [#862](#): Added support for declarative package config in a `setup.cfg` file.

## 7.120 v30.2.1

08 Dec 2016

- [#850](#): In test command, invoke `unittest.main` with indication not to exit the process.

## 7.121 v30.2.0

04 Dec 2016

- [#854](#): Bump to vendored [Packaging 16.8](#).

## 7.122 v30.1.0

03 Dec 2016

- [#846](#): Also trap ‘socket.error’ when opening URLs in `package_index`.
- [#849](#): Manifest processing now matches the filename pattern anywhere in the filename and not just at the start. Restores behavior found prior to 28.5.0.

## 7.123 v30.0.0

01 Dec 2016

- [#864](#): Drop support for Python 3.2. Systems requiring Python 3.2 support must use ‘setuptools < 30’.
- [#825](#): Suppress warnings for single files.
- [#830](#) via [#843](#): Once again restored inclusion of data files to sdist, but now trap `TypeError` caused by techniques employed `rjsmin` and similar.

## 7.124 v29.0.1

26 Nov 2016

- [#861](#): Re-release of v29.0.1 with the executable script launchers bundled. Now, launchers are included by default and users that want to disable this behavior must set the environment variable ‘`SETUP-TOOLS_INSTALL_WINDOWS_SPECIFIC_FILES`’ to a false value like “false” or “0”.

## 7.125 v29.0.0

25 Nov 2016

- [#841](#): Drop special exception for packages invoking `win32com` during the build/install process. See [Distribute #118](#) for history.

## 7.126 v28.8.0

04 Nov 2016

- [#629](#): Per the discussion, refine the sorting to use version value order for more accurate detection of the latest available version when scanning for packages. See also [#829](#).
- [#837](#): Rely on the config var “SO” for Python 3.3.0 only when determining the ext filename.



## 7.127 v28.7.1

29 Oct 2016

- [#827](#): Update PyPI root for dependency links.
- [#833](#): Backed out changes from [#830](#) as the implementation seems to have problems in some cases.

## 7.128 v28.7.0

28 Oct 2016

- [#832](#): Moved much of the namespace package handling functionality into a separate module for re-use in something like [#789](#).
- [#830](#): `sdist` command no longer suppresses the inclusion of data files, re-aligning with the expectation of `distutils` and addressing [#274](#) and [#521](#).

## 7.129 v28.6.1

19 Oct 2016

- [#816](#): Fix manifest file list order in tests.

## 7.130 v28.6.0

16 Oct 2016

- [#629](#): When scanning for packages, `pkg_resources` now ignores empty egg-info directories and gives precedence to packages whose versions are lexicographically greatest, a rough approximation for preferring the latest available version.

## 7.131 v28.5.0

14 Oct 2016

- [#810](#): Tests are now invoked with `tox` and not `setup.py` test.
- [#249](#) and [#450](#) via [#764](#): Avoid scanning the whole tree when building the manifest. Also fixes a long-standing bug where patterns in `MANIFEST.in` had implicit wildcard matching. This caused `global-exclude .foo` to exclude all `*.foo` files, but also `global-exclude bar.py` to exclude `foo_bar.py`.

## 7.132 v28.4.0

14 Oct 2016

- [#732](#): Now extras with a hyphen are honored per [PEP 426](#).
- [#811](#): Update to `pyparsing` 2.1.10.

- Updated `setuptools.command.sdist` to re-use most of the functionality directly from `distutils.command.sdist` for the `add_defaults` method with strategic overrides. See [#750](#) for rationale.
- [#760](#) via [#762](#): Look for certificate bundle where SUSE Linux typically presents it. Use `certifi.where()` to locate the bundle.

## 7.133 v28.3.0

07 Oct 2016

- [#809](#): In `find_packages()`, restore support for excluding a parent package without excluding a child package.
- [#805](#): Disable `-nspkg.pth` behavior on Python 3.3+ where [PEP-420](#) functionality is adequate. Fixes [pip #1924](#).

## 7.134 v28.1.0

01 Oct 2016

- [#803](#): Bump `certifi` to 2016.9.26.

## 7.135 v28.0.0

27 Sep 2016

- [#733](#): Do not search excluded directories for packages. This introduced a backwards incompatible change in `find_packages()` so that `find_packages(exclude=['foo']) == []`, excluding subpackages of `foo`. Previously, `find_packages(exclude=['foo']) == ['foo.bar']`, even though the parent `foo` package was excluded.
- [#795](#): Bump `certifi`.
- [#719](#): Suppress decoding errors and instead log a warning when metadata cannot be decoded.

## 7.136 v27.3.1

27 Sep 2016

- [#790](#): In MSVC monkeypatching, explicitly patch each function by name in the target module instead of inferring the module from the function's `__module__`. Improves compatibility with other packages that might have previously patched `distutils` functions (i.e. NumPy).

## 7.137 v27.3.0

20 Sep 2016

- [#794](#): In `test` command, add installed eggs to `PYTHONPATH` when invoking tests so that subprocesses will also have the dependencies available. Fixes [tox 330](#).
- [#795](#): Update vendored `pyparsing` 2.1.9.

## 7.138 v27.2.0

14 Sep 2016

- [#520](#) and [#513](#): Suppress `ValueErrors` in `fixup_namespace_packages` when lookup fails.
- Nicer, more consistent interfaces for `msvc` monkeypatching.

## 7.139 v27.1.2

09 Sep 2016

- [#779](#) via [#781](#): Fix circular import.

## 7.140 v27.1.1

09 Sep 2016

- [#778](#): Fix `MSVC` monkeypatching.

## 7.141 v27.1.0

09 Sep 2016

- Introduce the (private) `monkey` module to encapsulate the `distutils` monkeypatching behavior.

## 7.142 v27.0.0

09 Sep 2016

- Now use `Warehouse` by default for `upload`, patching `distutils.config.PyPIRCCommand` to affect default behavior.

Any config in `.pypirc` should be updated to replace

<https://pypi.python.org/pypi/>

with

<https://upload.pypi.org/legacy/>

Similarly, any passwords stored in the keyring should be updated to use this new value for “system”.

The `upload_docs` command will continue to use the `python.org` site, but the command is now deprecated. Users are urged to use `Read The Docs` instead.

- [#776](#): Use `EXT_SUFFIX` for `py_limited_api` renaming.
- [#774](#) and [#775](#): Use `LegacyVersion` from `packaging` when detecting `numpy` versions.

## 7.143 v26.1.1

29 Aug 2016

- Re-release of 26.1.0 with `pytest` pinned to allow for automated deployment and thus proper packaging environment variables, fixing issues with missing executable launchers.

## 7.144 v26.1.0

28 Aug 2016

- [#763](#): `pkg_resources.get_default_cache` now defers to the [appdirs](#) project to resolve the cache directory. Adds a vendored dependency on `appdirs` to `pkg_resources`.

## 7.145 v26.0.0

20 Aug 2016

- [#748](#): By default, sdist's are now produced in gzipped tarfile format by default on all platforms, adding forward compatibility for the same behavior in Python 3.6 (See [Python #27819](#)).
- [#459](#) via [#736](#): On Windows with script launchers, `sys.argv[0]` now reflects the name of the entry point, consistent with the behavior in `distlib` and `pip` wrappers.
- [#752](#) via [#753](#): When indicating `py_limited_api` to `Extension`, it must be passed as a keyword argument.

## 7.146 v25.4.0

19 Aug 2016

- Add `Extension(py_limited_api=True)`. When set to a truthy value, that extension gets a filename appropriate for code using `Py_LIMITED_API`. When used correctly this allows a single compiled extension to work on all future versions of CPython 3. The `py_limited_api` argument only controls the filename. To be compatible with multiple versions of Python 3, the C extension will also need to set `-DPy_LIMITED_API=...` and be modified to use only the functions in the limited API.

## 7.147 v25.3.0

19 Aug 2016

- [#739](#) Fix unquoted libpaths by fixing compatibility between `numpy.distutils` and `distutils._msvccompiler` for `numpy < 1.11.2` (Fix issue [#728](#), error also fixed in Numpy).
- [#731](#): Bump certifi.
- Style updates. See [#740](#), [#741](#), [#743](#), [#744](#), [#742](#), [#747](#).
- [#735](#): include license file.

## 7.148 v25.2.0

12 Aug 2016

- [#612](#) via [#730](#): Add a LICENSE file which needs to be provided by the terms of the MIT license.

## 7.149 v25.1.6

05 Aug 2016

- [#725](#): revert *library\_dir\_option* patch (Error is related to *numpy.distutils* and make errors on non Numpy users).

## 7.150 v25.1.5

05 Aug 2016

- [#720](#)
- [#723](#): Improve patch for *library\_dir\_option*.

## 7.151 v25.1.4

04 Aug 2016

- [#717](#)
- [#713](#)
- [#707](#): Fix Python 2 compatibility for MSVC by catching errors properly.
- [#715](#): Fix unquoted libpaths by patching *library\_dir\_option*.

## 7.152 v25.1.3

02 Aug 2016

- [#714](#) and [#704](#): Revert fix as it breaks other components downstream that can't handle unicode. See [#709](#), [#710](#), and [#712](#).

## 7.153 v25.1.2

01 Aug 2016

- [#704](#): Fix errors when installing a zip sdist that contained files named with non-ascii characters on Windows would crash the install when it attempted to clean up the build.
- [#646](#): MSVC compatibility - catch errors properly in RegistryInfo.lookup.
- [#702](#): Prevent UnboundLocalError when initial working\_set is empty.

## 7.154 v25.1.1

28 Jul 2016

- [#686](#): Fix issue in `sys.path` ordering by `pkg_resources` when rewrite technique is “raw”.
- [#699](#): Fix typo in `msvc` support.

## 7.155 v25.1.0

25 Jul 2016

- [#609](#): Setuptools will now try to download a distribution from the next possible download location if the first download fails. This means you can now specify multiple links as `dependency_links` and all links will be tried until a working download link is encountered.

## 7.156 v25.0.2

25 Jul 2016

- [#688](#): Fix `AttributeError` in `setup.py` when invoked not from the current directory.

## 7.157 v25.0.1

25 Jul 2016

- Cleanup of `setup.py` script.
- Fixed documentation builders by allowing `setup.py` to be imported without having bootstrapped the metadata.
- More style cleanup. See [#677](#), [#678](#), [#679](#), [#681](#), [#685](#).

## 7.158 v25.0.0

23 Jul 2016

- [#674](#): Default `sys.path` manipulation by `easy-install.pth` is now “raw”, meaning that when writing `easy-install.pth` during any install operation, the `sys.path` will not be rewritten and will no longer give preference to `easy_installed` packages.

To retain the old behavior when using any `easy_install` operation (including `setup.py install` when `setuptools` is present), set the environment variable:

`SETUPTOOLS_SYS_PATH_TECHNIQUE=rewrite`

This project hopes that that few if any environments find it necessary to retain the old behavior, and intends to drop support for it altogether in a future release. Please report any relevant concerns in the ticket for this change.

## 7.159 v24.3.1

23 Jul 2016

- [#398](#): Fix shebang handling on Windows in script headers where spaces in `sys.executable` would produce an improperly-formatted shebang header, introduced in 12.0 with the fix for [#188](#).
- [#663](#), [#670](#): More style updates.

## 7.160 v24.3.0

21 Jul 2016

- [#516](#): Disable `os.link` to avoid hard linking in `sdist.make_distribution`, avoiding errors on systems that support hard links but not on the file system in which the build is occurring.

## 7.161 v24.2.1

21 Jul 2016

- [#667](#): Update Metadata-Version to 1.2 when `python_requires` is supplied.

## 7.162 v24.2.0

20 Jul 2016

- [#631](#): Add support for `python_requires` keyword.

## 7.163 v24.1.1

20 Jul 2016

- More style updates. See [#660](#), [#661](#), [#641](#).

## 7.164 v24.1.0

20 Jul 2016

- [#659](#): `setup.py` now will fail fast and with a helpful error message when the necessary metadata is missing.
- More style updates. See [#656](#), [#635](#), [#640](#), [#644](#), [#650](#), [#652](#), and [#655](#).

## 7.165 v24.0.3

14 Jul 2016

- Updated style in much of the codebase to match community expectations. See [#632](#), [#633](#), [#634](#), [#637](#), [#639](#), [#638](#), [#642](#), [#648](#).

## 7.166 v24.0.2

04 Jul 2016

- If MSVC++14 is needed `setuptools.msvc` now redirect user to Visual C++ Build Tools web page.

## 7.167 v24.0.1

03 Jul 2016

- [#625](#) and [#626](#): Fixes on `setuptools.msvc` mainly for Python 2 and Linux.

## 7.168 v24.0.0

02 Jul 2016

- Pull Request [#174](#): Add more aggressive support for standalone Microsoft Visual C++ compilers in `msvc9compiler` patch. Particularly : Windows SDK 6.1 and 7.0 (MSVC++ 9.0), Windows SDK 7.1 (MSVC++ 10.0), Visual C++ Build Tools 2015 (MSVC++14)
- Renamed `setuptools.msvc9_support` to `setuptools.msvc`.

## 7.169 v23.2.1

02 Jul 2016

Re-release of v23.2.0, which was missing the intended commits.

- [#623](#): Remove used of deprecated ‘U’ flag when reading manifests.

## 7.170 v23.1.0

24 Jun 2016

- [#619](#): Deprecate `tag_svn_revision` distribution option.

## 7.171 v23.0.0

09 Jun 2016

- [#611](#): Removed ARM executables for CLI and GUI script launchers on Windows. If this was a feature you cared about, please comment in the ticket.
- [#604](#): Removed docs building support. The project now relies on documentation hosted at <https://setuptools.readthedocs.io/>.



## 7.172 v22.0.5

03 Jun 2016

- [#604](#): Restore repository for `upload_docs` command to restore publishing of docs during release.

## 7.173 v22.0.4

03 Jun 2016

- [#589](#): Upload releases to pypi.io using the upload hostname and legacy path.

## 7.174 v22.0.3

03 Jun 2016

- [#589](#): Releases are now uploaded to pypi.io (Warehouse) even when releases are made on Twine via Travis.

## 7.175 v22.0.2

03 Jun 2016

- [#589](#): Releases are now uploaded to pypi.io (Warehouse).

## 7.176 v22.0.1

03 Jun 2016

- [#190](#): On Python 2, if unicode is passed for packages to `build_py` command, it will be handled just as with text on Python 3.

## 7.177 v22.0.0

01 Jun 2016

Intended to be v21.3.0, but jaraco accidentally released as a major bump.

- [#598](#): Setuptools now lists itself first in the User-Agent for web requests, better following the guidelines in [RFC 7231](#).

## 7.178 v21.2.2

29 May 2016

- Minor fixes to changelog and docs.

## 7.179 v21.2.1

22 May 2016

- [#261](#): Exclude directories when resolving globs in `package_data`.

## 7.180 v21.2.0

21 May 2016

- [#539](#): In the `easy_install` `get_site_dirs`, honor all paths found in `site.getsitepackages`.

## 7.181 v21.1.0

18 May 2016

- [#572](#): In `build_ext`, now always import `_CONFIG_VARS` from `distutils` rather than from `sysconfig` to allow `distutils.sysconfig.customize_compiler` configure the OS X compiler for `-dynamiclib`.

## 7.182 v21.0.0

02 May 2016

- Removed `ez_setup.py` from Setuptools sdist. The bootstrap script will be maintained in its own branch and should be generally be retrieved from its canonical location at [https://bootstrap.pypa.io/ez\\_setup.py](https://bootstrap.pypa.io/ez_setup.py).

## 7.183 v20.10.0

25 Apr 2016

- [#553](#): `egg_info` section is now generated in a deterministic order, matching the order generated by earlier versions of Python. Except on Python 2.6, order is preserved when existing settings are present.
- [#556](#): Update to [Packaging 16.7](#), restoring support for deprecated `python_implementation` marker.
- [#555](#): Upload command now prompts for a password when uploading to PyPI (or other repository) if no password is present in `.pypirc` or in the keyring.

## 7.184 v20.9.0

16 Apr 2016

- [#548](#): Update certify version to 2016.2.28
- [#545](#): Safely handle deletion of non-zip eggs in `rotate` command.

## 7.185 v20.8.1

15 Apr 2016

- [Issue #544](#): Fix issue with extra environment marker processing in WorkingSet due to refactor in v20.7.0.

## 7.186 v20.8.0

15 Apr 2016

- [Issue #543](#): Re-release so that latest release doesn't cause déjà vu with distribute and setuptools 0.7 in older environments.

## 7.187 v20.7.0

10 Apr 2016

- Refactored extra environment marker processing in WorkingSet.
- [Issue #533](#): Fixed intermittent test failures.
- [Issue #536](#): In `msvc9_support`, trap additional exceptions that might occur when importing `distutils.msvc9compiler` in mingw environments.
- [Issue #537](#): Provide better context when package metadata fails to decode in UTF-8.

## 7.188 v20.6.8

09 May 2016

- [Issue #523](#): Restored support for environment markers, now honoring 'extra' environment markers.

## 7.189 v20.6.7

31 Mar 2016

- [Issue #523](#): Disabled support for environment markers introduced in v20.5.

## 7.190 v20.6.6

30 Mar 2016

- [Issue #503](#): Restore support for [PEP 345](#) environment markers by updating to [Packaging 16.6](#).

## 7.191 v20.6.0

29 Mar 2016

- New release process that relies on [bumpversion](#) and Travis CI for continuous deployment.
- Project versioning semantics now follow [semver](#) precisely. The ‘v’ prefix on version numbers now also allows version numbers to be referenced in the changelog, e.g. <http://setuptools.readthedocs.io/en/latest/history.html#v20-6-0>.

## 7.192 20.5

29 Mar 2016

- [BB Pull Request #185, #470](#): Add support for environment markers in requirements in `install_requires`, `setup_requires`, `tests_require` as well as adding a test for the existing `extra_requires` machinery.

## 7.193 20.4

29 Mar 2016

- [Issue #422](#): Moved hosting to [Github](#) from [Bitbucket](#). Issues have been migrated, though all issues and comments are attributed to `bb-migration`. So if you have a particular issue or issues to which you’ve been subscribed, you will want to “watch” the equivalent issue in Github. The Bitbucket project will be retained for the indefinite future, but Github now hosts the canonical project repository.

## 7.194 20.3.1

18 Mar 2016

- [Issue #519](#): Remove import hook when reloading the `pkg_resources` module.
- [BB Pull Request #184](#): Update documentation in `pkg_resources` around new `Requirement` implementation.

## 7.195 20.3

15 Mar 2016

- [BB Pull Request #179](#): `pkg_resources.Requirement` objects are now a subclass of `packaging.requirements.Requirement`, allowing any environment markers and url (if any) to be affiliated with the requirement
- [BB Pull Request #179](#): Restore use of `RequirementParseError` exception unintentionally dropped in 20.2.

## 7.196 20.2.2

27 Feb 2016

- [Issue #502](#): Correct regression in parsing of multiple version specifiers separated by commas and spaces.

## 7.197 20.2.1

24 Feb 2016

- [Issue #499](#): Restore compatibility for legacy versions by bumping to [packaging 16.4](#).

## 7.198 20.2

19 Feb 2016

- Changelog now includes release dates and links to PEPs.
- [BB Pull Request #173](#): Replace dual [PEP 345](#) `_markerlib` implementation and [PEP 426](#) implementation of environment marker support from [packaging 16.1](#) and [PEP 508](#). Fixes [Issue #122](#). See also [BB Pull Request #175](#), [BB Pull Request #168](#), and [BB Pull Request #164](#). Additionally:
  - `Requirement.parse` no longer retains the order of extras.
  - `parse_requirements` now requires that all versions be [PEP-440](#) compliant, as revealed in [#499](#). Packages released with invalid local versions should be re-released using the proper local version syntax, e.g. `mypkg-1.0+myorg.1`.

## 7.199 20.1.1

12 Feb 2016

- Update `upload_docs` command to also honor keyring for password resolution.

## 7.200 20.1

11 Feb 2016

- Added support for using passwords from keyring in the upload command. See [the upload docs](#) for details.

## 7.201 20.0

07 Feb 2016

- [Issue #118](#): Once again omit the package metadata (`egg-info`) from the list of outputs in `--record`. This version of setuptools can no longer be used to upgrade pip earlier than 6.0.

## 7.202 19.7

03 Feb 2016

- Off-project PR: [0dcee79](#) and [f9bd9b9](#) For FreeBSD, also [honor root certificates from ca\\_root\\_nss](#).

## 7.203 19.6.2

31 Jan 2016

- [Issue #491](#): Correct regression incurred in 19.4 where a double-namespace package installed using pip would cause a `TypeError`.

## 7.204 19.6.1

29 Jan 2016

- Restore compatibility for PyPy 3 compatibility lost in 19.4.1 addressing [Issue #487](#).
- `setuptools.launch` shim now loads scripts in a new namespace, avoiding getting relative imports from the setuptools package on Python 2.

## 7.205 19.6

24 Jan 2016

- Added a new entry script `setuptools.launch`, implementing the shim found in `pip.util.setuptools_build`. Use this command to launch distutils-only packages under setuptools in the same way that pip does, causing the setuptools monkeypatching of distutils to be invoked prior to invoking a script. Useful for debugging or otherwise installing a distutils-only package under setuptools when pip isn't available or otherwise does not expose the desired functionality. For example:

```
$ python -m setuptools.launch setup.py develop
```

- [Issue #488](#): Fix dual manifestation of Extension class in extension packages installed as dependencies when Cython is present.

## 7.206 19.5

23 Jan 2016

- [Issue #486](#): Correct `TypeError` when `getfilesystemencoding` returns `None`.
- [Issue #139](#): Clarified the license as MIT.
- [BB Pull Request #169](#): Removed special handling of command spec in scripts for Jython.

## 7.207 19.4.1

23 Jan 2016

- [Issue #487](#): Use direct invocation of `importlib.machinery` in `pkg_resources` to avoid missing detection on relevant platforms.

## 7.208 19.4

16 Jan 2016

- [Issue #341](#): Correct error in path handling of package data files in `build_py` command when package is empty.
- [Distribute #323](#), [Issue #141](#), [Issue #207](#), and [BB Pull Request #167](#): Another implementation of `pkg_resources.WorkingSet` and `pkg_resources.Distribution` that supports replacing an extant package with a new one, allowing for `setup_requires` dependencies to supersede installed packages for the session.

## 7.209 19.3

06 Jan 2016

- [Issue #229](#): Implement new technique for readily incorporating dependencies conditionally from vendored copies or primary locations. Adds a new dependency on `six`.

## 7.210 19.2

25 Dec 2015

- [BB Pull Request #163](#): Add `get_command_list` method to `Distribution`.
- [BB Pull Request #162](#): Add missing whitespace to multiline string literals.

## 7.211 19.1.1

16 Dec 2015

- [Issue #476](#): Cast version to string (using default encoding) to avoid creating Unicode types on Python 2 clients.
- [Issue #477](#): In Powershell downloader, use explicit rendering of strings, rather than rely on `repr`, which can be incorrect (especially on Python 2).

## 7.212 19.1

16 Dec 2015

- [Issue #215](#): The bootstrap script `ez_setup.py` now automatically detects the latest version of setuptools (using PyPI JSON API) rather than hard-coding a particular value.
- [Issue #475](#): Fix incorrect usage in `_translate_metadata2`.

## 7.213 19.0

15 Dec 2015

- [Issue #442](#): Use RawConfigParser for parsing .pyirc file. Interpolated values are no longer honored in .pyirc files.

## 7.214 18.8.1

13 Dec 2015

- [Issue #440](#): Prevent infinite recursion when a SandboxViolation or other UnpickleableException occurs in a sandbox context with setuptools hidden. Fixes regression introduced in Setuptools 12.0.

## 7.215 18.8

11 Dec 2015

- Deprecated `egg_info.get_pkg_info_revision`.
- [Issue #471](#): Don't rely on repr for an HTML attribute value in `package_index`.
- [Issue #419](#): Avoid errors in FileMetadata when the metadata directory is broken.
- [Issue #472](#): Remove deprecated use of 'U' in mode parameter when opening files.

## 7.216 18.7.1

01 Dec 2015

- [Issue #469](#): Refactored logic for [Issue #419](#) fix to re-use metadata loading from Provider.

## 7.217 18.7

28 Nov 2015

- Update dependency on `certify`.
- [BB Pull Request #160](#): Improve detection of gui script in `easy_install._adjust_header`.
- Made `test.test_args` a non-data property; alternate fix for the issue reported in [BB Pull Request #155](#).
- [Issue #453](#): In `ez_setup` bootstrap module, unload all `pkg_resources` modules following download.
- [BB Pull Request #158](#): Honor [PEP-488](#) when excluding files for namespace packages.
- [Issue #419](#) and [BB Pull Request #144](#): Add experimental support for reading the version info from `distutils`-installed metadata rather than using the version in the filename.

## 7.218 18.6.1

24 Nov 2015

- [Issue #464](#): Correct regression in invocation of superclass on old-style class on Python 2.



## 7.219 18.6

24 Nov 2015

- [Issue #439](#): When installing entry\_point scripts under development, omit the version number of the package, allowing any version of the package to be used.

## 7.220 18.5

01 Nov 2015

- In preparation for dropping support for Python 3.2, a warning is now logged when pkg\_resources is imported on Python 3.2 or earlier Python 3 versions.
- Add support for python\_platform\_implementation environment marker.
- Fix dictionary mutation during iteration.

## 7.221 18.4

10 Oct 2015

- [Issue #446](#): Test command now always invokes unittest, even if no test suite is supplied.

## 7.222 18.3.2

19 Sep 2015

- Correct another regression in setuptools.findall where the fix for [Python #12885](#) was lost.

## 7.223 18.3.1

07 Sep 2015

- [Issue #425](#): Correct regression in setuptools.findall.

## 7.224 18.3

06 Sep 2015

- [BB Pull Request #135](#): Setuptools now allows disabling of the manipulation of the sys.path during the processing of the easy-install.pth file. To do so, set the environment variable SETUPTOOLS\_SYS\_PATH\_TECHNIQUE to anything but “rewrite” (consider “raw”). During any install operation with manipulation disabled, setuptools packages will be appended to sys.path naturally.

Future versions may change the default behavior to disable manipulation. If so, the default behavior can be retained by setting the variable to “rewrite”.

- [Issue #257](#): `easy_install --version` now shows more detail about the installation location and Python version.

- Refactor `setuptools.findall` in preparation for re-submission back to distutils.

## 7.225 18.2

19 Aug 2015

- [Issue #412](#): More efficient directory search in `find_packages`.

## 7.226 18.1

02 Aug 2015

- Upgrade to vendored [packaging 15.3](#).

## 7.227 18.0.1

24 Jun 2015

- [Issue #401](#): Fix failure in test suite.

## 7.228 18.0

13 Jun 2015

- Dropped support for builds with Pyrex. Only Cython is supported.
- [Issue #288](#): Detect Cython later in the build process, after `setup_requires` dependencies are resolved. Projects backed by Cython can now be readily built with a `setup_requires` dependency. For example:

```
ext = setuptools.Extension('mylib', ['src/CythonStuff.pyx', 'src/CStuff.c'])
setuptools.setup(
    ...
    ext_modules=[ext],
    setup_requires=['cython'],
)
```

For compatibility with older versions of setuptools, packagers should still include `src/CythonMod.c` in the source distributions or require that Cython be present before building source distributions. However, for systems with this build of setuptools, Cython will be downloaded on demand.

- [Issue #396](#): Fixed test failure on OS X.
- [BB Pull Request #136](#): Remove excessive quoting from shebang headers for Jython.

## 7.229 17.1.1

08 Jun 2015

- Backed out unintended changes to `pkg_resources`, restoring removal of deprecated `imp` module ([ref](#)).

## 7.230 17.1

07 Jun 2015

- [Issue #380](#): Add support for range operators on environment marker evaluation.

## 7.231 17.0

28 May 2015

- [Issue #378](#): Do not use internal `importlib._bootstrap` module.
- [Issue #390](#): Disallow console scripts with path separators in the name. Removes unintended functionality and brings behavior into parity with pip.

## 7.232 16.0

18 May 2015

- [BB Pull Request #130](#): Better error messages for errors in parsed requirements.
- [BB Pull Request #133](#): Removed `setuptools.tests` from the installed packages.
- [BB Pull Request #129](#): Address deprecation warning due to usage of `imp` module.

## 7.233 15.2

26 Apr 2015

- [Issue #373](#): Provisionally expose `pkg_resources._initialize_master_working_set`, allowing for imperative re-initialization of the master working set.

## 7.234 15.1

15 Apr 2015

- Updated to [Packaging 15.1](#) to address [Packaging #28](#).
- Fix `setuptools.sandbox._execfile()` with Python 3.1.

## 7.235 15.0

03 Apr 2015

- [BB Pull Request #126](#): `DistributionNotFound` message now lists the package or packages that required it. E.g.:

```
pkg_resources.DistributionNotFound: The 'colorama>=0.3.1' distribution was not
↪ found and is required by smlib.log.
```

Note that `zc.buildout` once depended on the string rendering of this message to determine the package that was not found. This expectation has since been changed, but older versions of buildout may experience problems. See [Buildout #242](#) for details.

## 7.236 14.3.1

20 Mar 2015

- [Issue #307](#): Removed [PEP-440](#) warning during parsing of versions in `pkg_resources.Distribution`.
- [Issue #364](#): Replace deprecated usage with recommended usage of `EntryPoint.load`.

## 7.237 14.3

15 Mar 2015

- [Issue #254](#): When creating temporary egg cache on Unix, use mode 755 for creating the directory to avoid the subsequent warning if the directory is group writable.

## 7.238 14.2

15 Mar 2015

- [Issue #137](#): Update `Distribution.hashcmp` so that Distributions with `None` for `pyversion` or `platform` can be compared against Distributions defining those attributes.

## 7.239 14.1.1

14 Mar 2015

- [Issue #360](#): Removed undesirable behavior from test runs, preventing write tests and installation to system site packages.

## 7.240 14.1

14 Mar 2015

- [BB Pull Request #125](#): Add `__ne__` to `Requirement` class.
- Various refactoring of `easy_install`.

## 7.241 14.0

06 Mar 2015

- Bootstrap script now accepts `--to-dir` to customize save directory or allow for re-use of existing repository of setuptools versions. See [BB Pull Request #112](#) for background.

- [Issue #285](#): `easy_install` no longer will default to installing packages to the “user site packages” directory if it is itself installed there. Instead, the user must pass `--user` in all cases to install packages to the user site packages. This behavior now matches that of “pip install”. To configure an environment to always install to the user site packages, consider using the “install-dir” and “scripts-dir” parameters to `easy_install` through an appropriate `distutils` config file.

## 7.242 13.0.2

06 Mar 2015

- [Issue #359](#): Include `pytest.ini` in the `sdist` so invocation of `py.test` on the `sdist` honors the `pytest` configuration.

## 7.243 13.0.1

05 Mar 2015

Re-release of 13.0. Intermittent connectivity issues caused the release process to fail and PyPI uploads no longer accept files for 13.0.

## 7.244 13.0

05 Mar 2015

- [Issue #356](#): Back out [BB Pull Request #119](#) as it requires Setuptools 10 or later as the source during an upgrade.
- Removed `build_py` class from `setup.py`. According to 892f439d216e, this functionality was added to support upgrades from old `Distribute` versions, 0.6.5 and 0.6.6.

## 7.245 12.4

04 Mar 2015

- [BB Pull Request #119](#): Restore writing of `setup_requires` to metadata (previously added in 8.4 and removed in 9.0).

## 7.246 12.3

26 Feb 2015

- Documentation is now linked using the `rst.linker` package.
- Fix `setuptools.command.easy_install.extract_wininst_cfg()` with Python 2.6 and 2.7.
- [Issue #354](#). Added documentation on building setuptools documentation.

## 7.247 12.2

16 Feb 2015

- [Issue #345](#): Unload all modules under `pkg_resources` during `ez_setup.use_setuptools()`.
- [Issue #336](#): Removed deprecation from `ez_setup.use_setuptools`, as it is clearly still used by buildout's bootstrap. `ez_setup` remains deprecated for use by individual packages.
- Simplified implementation of `ez_setup.use_setuptools`.

## 7.248 12.1

10 Feb 2015

- [BB Pull Request #118](#): Soften warning for non-normalized versions in Distribution.

## 7.249 12.0.5

26 Jan 2015

- [Issue #339](#): Correct Attribute reference in `cant_write_to_target`.
- [Issue #336](#): Deprecated `ez_setup.use_setuptools`.

## 7.250 12.0.4

20 Jan 2015

- [Issue #335](#): Fix script header generation on Windows.

## 7.251 12.0.3

18 Jan 2015

- Fixed incorrect class attribute in `install_scripts`. Tests would be nice.

## 7.252 12.0.2

18 Jan 2015

- [Issue #331](#): Fixed `install_scripts` command on Windows systems corrupting the header.

## 7.253 12.0.1

16 Jan 2015

- Restore `setuptools.command.easy_install.sys_executable` for pbr compatibility. For the future, tools should construct a `CommandSpec` explicitly.

## 7.254 12.0

16 Jan 2015

- [Issue #188](#): Setuptools now support multiple entities in the value for `build.executable`, such that an executable of “`/usr/bin/env my-python`” may be specified. This means that systems with a specified executable whose name has spaces in the path must be updated to escape or quote that value.
- Deprecated `easy_install.ScriptWriter.get_writer`, replaced by `.best()` with slightly different semantics (no `force_windows` flag).

## 7.255 11.3.1

06 Jan 2015

- [Issue #327](#): Formalize and restore support for any printable character in an entry point name.

## 7.256 11.3

05 Jan 2015

- Expose `EntryPoint.resolve` in place of `EntryPoint._load`, implementing the simple, non-requiring load. Deprecated all uses of `EntryPoint._load` except for calling with no parameters, which is just a shortcut for `ep.require(); ep.resolve();`.

Apps currently invoking `ep.load(require=False)` should instead do the following if wanting to avoid the deprecating warning:

```
getattr(ep, "resolve", lambda: ep.load(require=False))()
```

## 7.257 11.2

05 Jan 2015

- [Pip #2326](#): Report deprecation warning at stacklevel 2 for easier diagnosis.

## 7.258 11.1

04 Jan 2015

- [Issue #281](#): Since Setuptools 6.1 ([Issue #268](#)), a `ValueError` would be raised in certain cases where `VersionConflict` was raised with two arguments, which occurred in `pkg_resources.WorkingSet.find`. This release adds support for indicating the dependent packages while maintaining support for a `VersionConflict` when no dependent package context is known. New unit tests now capture the expected interface.

## 7.259 11.0

02 Jan 2015

- [Interop #3](#): Upgrade to [Packaging 15.0](#); updates to [PEP 440](#) so that >1.7 does not exclude 1.7.1 but does exclude 1.7.0 and 1.7.0.post1.

## 7.260 10.2.1

02 Jan 2015

- [Issue #323](#): Fix regression in entry point name parsing.

## 7.261 10.2

02 Jan 2015

- Deprecated use of `EntryPoint.load(require=False)`. Passing a boolean to a function to select behavior is an anti-pattern. Instead use `Entrypoint._load()`.
- Substantial refactoring of all unit tests. Tests are now much leaner and re-use a lot of fixtures and contexts for better clarity of purpose.

## 7.262 10.1

31 Dec 2014

- [Issue #320](#): Added a compatibility implementation of `sdist._default_revctrl` so that systems relying on that interface do not fail (namely, Ubuntu 12.04 and similar Debian releases).

## 7.263 10.0.1

30 Dec 2014

- [Issue #319](#): Fixed issue installing pure distutils packages.

## 7.264 10.0

30 Dec 2014

- [Issue #313](#): Removed built-in support for subversion. Projects wishing to retain support for subversion will need to use a third party library. The extant implementation is being ported to [setuptools\\_svn](#).
- [Issue #315](#): Updated setuptools to hide its own loaded modules during installation of another package. This change will enable setuptools to upgrade (or downgrade) itself even when its own metadata and implementation change.



## 7.265 9.1

29 Dec 2014

- Prefer vendored packaging library [as recommended](#).

## 7.266 9.0.1

29 Dec 2014

- [Issue #312](#): Restored presence of `pkg_resources` API tests (doctest) to `sdist`.

## 7.267 9.0

28 Dec 2014

- [Issue #314](#): Disabled support for `setup_requires` metadata to avoid issue where Setuptools was unable to upgrade over earlier versions.

## 7.268 8.4

26 Dec 2014

- [BB Pull Request #106](#): Now write `setup_requires` metadata.

## 7.269 8.3

24 Dec 2014

- [Issue #311](#): Decoupled `pkg_resources` from setuptools once again. `pkg_resources` is now a package instead of a module.

## 7.270 8.2.1

18 Dec 2014

- [Issue #306](#): Suppress warnings about Version format except in select scenarios (such as installation).

## 7.271 8.2

18 Dec 2014

- [BB Pull Request #85](#): Search egg-base when adding egg-info to manifest.

## 7.272 8.1

18 Dec 2014

- Upgrade `packaging` to 14.5, giving preference to “rc” as designator for release candidates over “c”.
- [PEP-440](#) warnings are now raised as their own class, `pkg_resources.PEP440Warning`, instead of `RuntimeWarning`.
- Disabled warnings on empty versions.

## 7.273 8.0.4

15 Dec 2014

- Upgrade `packaging` to 14.4, fixing an error where there is a different result for if 2.0.5 is contained within `>2.0dev` and `>2.0.dev` even though normalization rules should have made them equal.
- [Issue #296](#): Add warning when a version is parsed as legacy. This warning will make it easier for developers to recognize deprecated version numbers.

## 7.274 8.0.3

15 Dec 2014

- [Issue #296](#): Restored support for `__hash__` on `parse_version` results.

## 7.275 8.0.2

14 Dec 2014

- [Issue #296](#): Restored support for `__getitem__` and sort operations on `parse_version` result.

## 7.276 8.0.1

13 Dec 2014

- [Issue #296](#): Restore support for iteration over `parse_version` result, but deprecated that usage with a warning. Fixes failure with buildout.

## 7.277 8.0

13 Dec 2014

- Implement [PEP 440](#) within `pkg_resources` and `setuptools`. This change deprecates some version numbers such that they will no longer be installable without using the `==` escape hatch. See [the changes to test\\_resources](#) for specific examples of version numbers and specifiers that are no longer supported. Setuptools now “vendors” the `packaging` library.

## 7.278 7.0

19 Oct 2014

- [Issue #80](#), [Issue #209](#): Eggs that are downloaded for `setup_requires`, `test_requires`, etc. are now placed in a `./eggs` directory instead of directly in the current directory. This choice of location means the files can be readily managed (removed, ignored). Additionally, later phases or invocations of setuptools will not detect the package as already installed and ignore it for permanent install (See [#209](#)).

This change is indicated as backward-incompatible as installations that depend on the installation in the current directory will need to account for the new location. Systems that ignore `*.egg` will probably need to be adapted to ignore `.eggs`. The files will need to be manually moved or will be retrieved again. Most use cases will require no attention.

## 7.279 6.1

11 Oct 2014

- [Issue #268](#): When resolving package versions, a `VersionConflict` now reports which package previously required the conflicting version.

## 7.280 6.0.2

29 Sep 2014

- [Issue #262](#): Fixed regression in pip install due to egg-info directories being omitted. Re-opens [Issue #118](#).

## 7.281 6.0.1

27 Sep 2014

- [Issue #259](#): Fixed regression with namespace package handling on single version, externally managed installs.

## 7.282 6.0

27 Sep 2014

- [Issue #100](#): When building a distribution, Setuptools will no longer match default files using platform-dependent case sensitivity, but rather will only match the files if their case matches exactly. As a result, on Windows and other case-insensitive file systems, files with names such as `'readme.txt'` or `'README.TXT'` will be omitted from the distribution and a warning will be issued indicating that `'README.txt'` was not found. Other filenames affected are:
  - `README.rst`
  - `README`
  - `setup.cfg`
  - `setup.py` (or the script name)

- test/test\*.py

Any users producing distributions with filenames that match those above case-insensitively, but not case-sensitively, should rename those files in their repository for better portability.

- [BB Pull Request #72](#): When using `single_version_externally_managed`, the exclusion list now includes Python 3.2 `__pycache__` entries.
- [BB Pull Request #76](#) and [BB Pull Request #78](#): lines in `top_level.txt` are now ordered deterministically.
- [Issue #118](#): The egg-info directory is now no longer included in the list of outputs.
- [Issue #258](#): Setuptools now patches `distutils.msvc9compiler` to recognize the specially-packaged compiler package for easy extension module support on Python 2.6, 2.7, and 3.2.

## 7.283 5.8

18 Sep 2014

- [Issue #237](#): `pkg_resources` now uses explicit detection of Python 2 vs. Python 3, supporting environments where builtins have been patched to make Python 3 look more like Python 2.

## 7.284 5.7

15 Aug 2014

- [Issue #240](#): Based on real-world performance measures against 5.4, zip manifests are now cached in all circumstances. The `PKG_RESOURCES_CACHE_ZIP_MANIFESTS` environment variable is no longer relevant. The observed “memory increase” referenced in the 5.4 release notes and detailed in [Issue #154](#) was likely not an increase over the status quo, but rather only an increase over not storing the zip info at all.

## 7.285 5.6

14 Aug 2014

- [Issue #242](#): Use absolute imports in `svn_utils` to avoid issues if the installing package adds an xml module to the path.

## 7.286 5.5.1

10 Aug 2014

- [Issue #239](#): Fix typo in 5.5 such that fix did not take.

## 7.287 5.5

10 Aug 2014

- [Issue #239](#): Setuptools now includes the `setup_requires` directive on `Distribution` objects and validates the syntax just like `install_requires` and `tests_require` directives.

## 7.288 5.4.2

01 Aug 2014

- [Issue #236](#): Corrected regression in execfile implementation for Python 2.6.

## 7.289 5.4.1

06 Jul 2014

- [Python #7776](#): (ssl\_support) Correct usage of host for validation when tunneling for HTTPS.

## 7.290 5.4

05 Jul 2014

- [Issue #154](#): `pkg_resources` will now cache the zip manifests rather than re-processing the same file from disk multiple times, but only if the environment variable `PKG_RESOURCES_CACHE_ZIP_MANIFESTS` is set. Clients that package many modules in the same zip file will see some improvement in startup time by enabling this feature. This feature is not enabled by default because it causes a substantial increase in memory usage.

## 7.291 5.3

28 Jun 2014

- [Issue #185](#): Make svn tagging work on the new style SVN metadata. Thanks cazabon!
- Prune revision control directories (e.g .svn) from base path as well as sub-directories.

## 7.292 5.2

22 Jun 2014

- Added a [Developer Guide](#) to the official documentation.
- Some code refactoring and cleanup was done with no intended behavioral changes.
- During `install_egg_info`, the generated lines for namespace package `.pth` files are now processed even during a dry run.

## 7.293 5.1

15 Jun 2014

- [Issue #202](#): Implemented more robust cache invalidation for the `ZipImporter`, building on the work in [Issue #168](#). Special thanks to Jurko Gospodnetic and PJE.

## 7.294 5.0.2

15 Jun 2014

- [Issue #220](#): Restored script templates.

## 7.295 5.0.1

14 Jun 2014

- Renamed script templates to end with `.tmpl` now that they no longer need to be processed by 2to3. Fixes spurious syntax errors during build/install.

## 7.296 5.0

14 Jun 2014

- [Issue #218](#): Re-release of 3.8.1 to signal that it supersedes 4.x.
- Incidentally, script templates were updated not to include the triple-quote escaping.

## 7.297 3.7.1 and 3.8.1 and 4.0.1

- [Issue #213](#): Use legacy StringIO behavior for compatibility under pbr.
- [Issue #218](#): Setuptools 3.8.1 superseded 4.0.1, and 4.x was removed from the available versions to install.

## 7.298 4.0

01 Jun 2014

- [Issue #210](#): `setup.py develop` now copies scripts in binary mode rather than text mode, matching the behavior of the `install` command.

## 7.299 3.8

01 Jun 2014

- Extend [Issue #197](#) workaround to include all Python 3 versions prior to 3.2.2.

## 7.300 3.7

28 May 2014

- [Issue #193](#): Improved handling of Unicode filenames when building manifests.

## 7.301 3.6

07 May 2014

- [Issue #203](#): Honor proxy settings for Powershell downloader in the bootstrap routine.

## 7.302 3.5.2

07 May 2014

- [Issue #168](#): More robust handling of replaced zip files and stale caches. Fixes `ZipImportError` complaining about a ‘bad local header’.

## 7.303 3.5.1

04 May 2014

- [Issue #199](#): Restored `install._install` for compatibility with earlier NumPy versions.

## 7.304 3.5

03 May 2014

- [Issue #195](#): Follow symbolic links in `find_packages` (restoring behavior broken in 3.4).
- [Issue #197](#): On Python 3.1, `PKG-INFO` is now saved in a UTF-8 encoding instead of `sys.getpreferredencoding` to match the behavior on Python 2.6-3.4.
- [Issue #192](#): Preferred bootstrap location is now [https://bootstrap.pypa.io/ez\\_setup.py](https://bootstrap.pypa.io/ez_setup.py) (mirrored from former location).

## 7.305 3.4.4

11 Apr 2014

- [Issue #184](#): Correct failure where `find_package` over-matched packages when directory traversal isn’t short-circuited.

## 7.306 3.4.3

07 Apr 2014

- [Issue #183](#): Really fix test command with Python 3.1.

## 7.307 3.4.2

06 Apr 2014

- [Issue #183](#): Fix additional regression in test command on Python 3.1.

## 7.308 3.4.1

30 Mar 2014

- [Issue #180](#): Fix regression in test command not caught by `py.test-run` tests.

## 7.309 3.4

30 Mar 2014

- [Issue #176](#): Add parameter to the test command to support a custom test runner: `--test-runner` or `-r`.
- [Issue #177](#): Now assume most common invocation to install command on platforms/environments without stack support (issuing a warning). Setuptools now installs naturally on IronPython. Behavior on CPython should be unchanged.

## 7.310 3.3

16 Mar 2014

- Add `include` parameter to `setuptools.find_packages()`.

## 7.311 3.2

14 Mar 2014

- [BB Pull Request #39](#): Add support for C++ targets from Cython `.pyx` files.
- [Issue #162](#): Update dependency on `certifi` to 1.0.1.
- [Issue #164](#): Update dependency on `wincertstore` to 0.2.

## 7.312 3.1

08 Mar 2014

- [Issue #161](#): Restore Features functionality to allow backward compatibility (for Features) until the uses of that functionality is sufficiently removed.



## 7.313 3.0.2

06 Mar 2014

- Correct typo in previous bugfix.

## 7.314 3.0.1

06 Mar 2014

- [Issue #157](#): Restore support for Python 2.6 in bootstrap script where `zipfile.ZipFile` does not yet have support for context managers.

## 7.315 3.0

04 Mar 2014

- [Issue #125](#): Prevent Subversion support from creating a `~/subversion` directory just for checking the presence of a Subversion repository.
- [Issue #12](#): Namespace packages are now imported lazily. That is, the mere declaration of a namespace package in an egg on `sys.path` no longer causes it to be imported when `pkg_resources` is imported. Note that this change means that all of a namespace package's `__init__.py` files must include a `declare_namespace()` call in order to ensure that they will be handled properly at runtime. In 2.x it was possible to get away without including the declaration, but only at the cost of forcing namespace packages to be imported early, which 3.0 no longer does.
- [Issue #148](#): When building (bdist\_egg), setuptools no longer adds `__init__.py` files to namespace packages. Any packages that rely on this behavior will need to create `__init__.py` files and include the `declare_namespace()`.
- [Issue #7](#): Setuptools itself is now distributed as a zip archive in addition to tar archive. `ez_setup.py` now uses zip archive. This approach avoids the potential security vulnerabilities presented by use of tar archives in `ez_setup.py`. It also leverages the security features added to `ZipFile.extract` in Python 2.7.4.
- [Issue #65](#): Removed deprecated Features functionality.
- [BB Pull Request #28](#): Remove backport of `_bytecode_filenames` which is available in Python 2.6 and later, but also has better compatibility with Python 3 environments.
- [Issue #156](#): Fix spelling of `__PYENVV_LAUNCHER__` variable.

## 7.316 2.2

07 Feb 2014

- [Issue #141](#): Restored fix for allowing `setup_requires` dependencies to override installed dependencies during setup.
- [Issue #128](#): Fixed issue where only the first dependency link was honored in a distribution where multiple dependency links were supplied.

## 7.317 2.1.2

05 Feb 2014

- [Issue #144](#): Read `long_description` using `codecs` module to avoid errors installing on systems where `LANG=C`.

## 7.318 2.1.1

05 Feb 2014

- [Issue #139](#): Fix regression in `re_finder` for CVS repos (and maybe Git repos as well).

## 7.319 2.1

07 Jan 2014

- [Issue #129](#): Suppress inspection of `*.whl` files when searching for files in a zip-imported file.
- [Issue #131](#): Fix `RuntimeError` when constructing an egg fetcher.

## 7.320 2.0.2

29 Dec 2013

- Fix `NameError` during installation with Python implementations (e.g. Jython) not containing parser module.
- Fix `NameError` in `sdist:re_finder`.

## 7.321 2.0.1

15 Dec 2013

- [Issue #124](#): Fixed error in list detection in `upload_docs`.

## 7.322 2.0

07 Dec 2013

- [Issue #121](#): Exempt `lib2to3` pickled grammars from `DirectorySandbox`.
- [Issue #41](#): Dropped support for Python 2.4 and Python 2.5. Clients requiring setuptools for those versions of Python should use setuptools 1.x.
- Removed `setuptools.command.easy_install.HAS_USER_SITE`. Clients expecting this boolean variable should use `site.ENABLE_USER_SITE` instead.
- Removed `pkg_resources.ImpWrapper`. Clients that expected this class should use `pkgutil.ImpImporter` instead.

## 7.323 1.4.2

01 Dec 2013

- [Issue #116](#): Correct `TypeError` when reading a local package index on Python 3.

## 7.324 1.4.1

23 Nov 2013

- [Issue #114](#): Use `sys.getfilesystemencoding` for decoding config in `bdist_wininst` distributions.
- [Issue #105](#) and [Issue #113](#): Establish a more robust technique for determining the terminal encoding:

```
1. Try ``getpreferredencoding``
2. If that returns US_ASCII or None, try the encoding from
   ``getdefaultlocale``. If that encoding was a "fallback" because Python
   could not figure it out from the environment or OS, encoding remains
   unresolved.
3. If the encoding is resolved, then make sure Python actually implements
   the encoding.
4. On the event of an error or unknown codec, revert to fallbacks
   (UTF-8 on Darwin, ASCII on everything else).
5. On the encoding is 'mac-roman' on Darwin, use UTF-8 as 'mac-roman' was
   a bug on older Python releases.
```

On a side note, it would seem that the encoding only matters for when SVN does not yet support `--xml` and when getting repository and svn version numbers. The `--xml` technique should yield UTF-8 according to some messages on the SVN mailing lists. So if the version numbers are always 7-bit ASCII clean, it may be best to only support the file parsing methods for legacy SVN releases and support for SVN without the subprocess command would simply go away as support for the older SVN does.

## 7.325 1.4

17 Nov 2013

- [Issue #27](#): `easy_install` will now use credentials from `.pyirc` if present for connecting to the package index.
- [BB Pull Request #21](#): Omit unwanted newlines in `package_index._encode_auth` when the username/password pair length indicates wrapping.

## 7.326 1.3.2

09 Nov 2013

- [Issue #99](#): Fix filename encoding issues in SVN support.

## 7.327 1.3.1

07 Nov 2013

- Remove exuberant warning in SVN support when SVN is not used.

## 7.328 1.3

03 Nov 2013

- Address security vulnerability in SSL `match_hostname` check as reported in [Python #17997](#).
- Prefer `backports.ssl_match_hostname` for backport implementation if present.
- Correct `NameError` in `ssl_support` module (`socket.error`).

## 7.329 1.2

02 Nov 2013

- [Issue #26](#): Add support for SVN 1.7. Special thanks to Philip Thiem for the contribution.
- [Issue #93](#): Wheels are now distributed with every release. Note that as reported in [Issue #108](#), as of Pip 1.4, scripts aren't installed properly from wheels. Therefore, if using Pip to install setuptools from a wheel, the `easy_install` command will not be available.
- Setuptools “natural” launcher support, introduced in 1.0, is now officially supported.

## 7.330 1.1.7

11 Apr 2013

- Fixed behavior of `NameError` handling in ‘script template (dev).py’ (script launcher for ‘develop’ installs).
- `ez_setup.py` now ensures partial downloads are cleaned up following a failed download.
- [Distribute #363](#) and [Issue #55](#): Skip an `sdist` test that fails on locales other than UTF-8.

## 7.331 1.1.6

18 Sep 2013

- [Distribute #349](#): `sandbox.execfile` now opens the target file in binary mode, thus honoring a BOM in the file when compiled.

## 7.332 1.1.5

12 Sep 2013

- [Issue #69](#): Second attempt at fix (logic was reversed).

## 7.333 1.1.4

07 Sep 2013

- [Issue #77](#): Fix error in upload command (Python 2.4).

## 7.334 1.1.3

06 Sep 2013

- Fix `NameError` in previous patch.

## 7.335 1.1.2

06 Sep 2013

- [Issue #69](#): Correct issue where 404 errors are returned for URLs with fragments in them (such as `#egg=`).

## 7.336 1.1.1

03 Sep 2013

- [Issue #75](#): Add `--insecure` option to `ez_setup.py` to accommodate environments where a trusted SSL connection cannot be validated.
- [Issue #76](#): Fix `AttributeError` in upload command with Python 2.4.

## 7.337 1.1

26 Aug 2013

- [Issue #71 \(Distribute #333\)](#): `EasyInstall` now puts less emphasis on the condition when a host is blocked via `--allow-hosts`.
- [Issue #72](#): Restored Python 2.4 compatibility in `ez_setup.py`.

## 7.338 1.0

17 Aug 2013

- [Issue #60](#): On Windows, Setuptools supports deferring to another launcher, such as Vinay Sajip’s [pylauncher](#) (included with Python 3.3) to launch console and GUI scripts and not install its own launcher executables. This experimental functionality is currently only enabled if the `SETUPTOOLS_LAUNCHER` environment variable is set to “natural”. In the future, this behavior may become default, but only after it has matured and seen substantial adoption. The `SETUPTOOLS_LAUNCHER` also accepts “executable” to force the default behavior of creating launcher executables.
- [Issue #63](#): Bootstrap script (`ez_setup.py`) now prefers Powershell, `curl`, or `wget` for retrieving the Setuptools tarball for improved security of the install. The script will still fall back to a simple `urlopen` on platforms that do not have these tools.

- [Issue #65](#): Deprecated the `Features` functionality.
- [Issue #52](#): In `VerifyingHTTPSConn`, handle a tunnelled (proxied) connection.

### 7.338.1 Backward-Incompatible Changes

This release includes a couple of backward-incompatible changes, but most if not all users will find 1.0 a drop-in replacement for 0.9.

- [Issue #50](#): Normalized API of environment marker support. Specifically, removed line number and filename from `SyntaxErrors` when returned from `pkg_resources.invalid_marker`. Any clients depending on the specific string representation of exceptions returned by that function may need to be updated to account for this change.
- [Issue #50](#): `SyntaxErrors` generated by `pkg_resources.invalid_marker` are normalized for cross-implementation consistency.
- Removed `--ignore-conflicts-at-my-risk` and `--delete-conflicting` options to `easy_install`. These options have been deprecated since 0.6a11.

## 7.339 0.9.8

25 Jul 2013

- [Issue #53](#): Fix `NameErrors` in `_vcs_split_rev_from_url`.

## 7.340 0.9.7

22 Jul 2013

- [Issue #49](#): Correct `AttributeError` on PyPy where a `hashlib.HASH` object does not have a `.name` attribute.
- [Issue #34](#): Documentation now refers to bootstrap script in code repository referenced by bookmark.
- Add underscore-separated keys to environment markers (`markerlib`).

## 7.341 0.9.6

17 Jul 2013

- [Issue #44](#): Test failure on Python 2.4 when MD5 hash doesn't have a `.name` attribute.

## 7.342 0.9.5

15 Jul 2013

- [Python #17980](#): Fix security vulnerability in SSL certificate validation.

## 7.343 0.9.4

15 Jul 2013

- [Issue #43](#): Fix issue (introduced in 0.9.1) with version resolution when upgrading over other releases of Setup-tools.

## 7.344 0.9.3

15 Jul 2013

- [Issue #42](#): Fix new `AttributeError` introduced in last fix.

## 7.345 0.9.2

15 Jul 2013

- [Issue #42](#): Fix regression where blank checksums would trigger an `AttributeError`.

## 7.346 0.9.1

13 Jul 2013

- [Distribute #386](#): Allow other positional and keyword arguments to `os.open`.
- Corrected dependency on `certifi` mis-referenced in 0.9.

## 7.347 0.9

13 Jul 2013

- `package_index` now validates hashes other than MD5 in download links.

## 7.348 0.8

05 Jul 2013

- Code base now runs on Python 2.4 - Python 3.3 without Python 2to3 conversion.

## 7.349 0.7.8

04 Jul 2013

- [Distribute #375](#): Yet another fix for yet another regression.

## 7.350 0.7.7

02 Jul 2013

- [Distribute #375](#): Repair `AttributeError` created in last release (redo).
- [Issue #30](#): Added test for `get_cache_path`.

## 7.351 0.7.6

02 Jul 2013

- [Distribute #375](#): Repair `AttributeError` created in last release.

## 7.352 0.7.5

29 Jun 2013

- [Issue #21](#): Restore Python 2.4 compatibility in `test_easy_install`.
- [Distribute #375](#): Merged additional warning from `Distribute 0.6.46`.
- Now honor the environment variable `SETUPTOOLS_DISABLE_VERSIONED_EASY_INSTALL_SCRIPT` in addition to the now deprecated `DISTRIBUTE_DISABLE_VERSIONED_EASY_INSTALL_SCRIPT`.

## 7.353 0.7.4

19 Jun 2013

- [Issue #20](#): Fix comparison of parsed SVN version on Python 3.

## 7.354 0.7.3

18 Jun 2013

- [Issue #1](#): Disable installation of Windows-specific files on non-Windows systems.
- Use new `sysconfig` module with Python 2.7 or  $\geq 3.2$ .

## 7.355 0.7.2

09 Jun 2013

- [Issue #14](#): Use `markerlib` when the `parser` module is not available.
- [Issue #10](#): `ez_setup.py` now uses HTTPS to download setuptools from PyPI.



## 7.356 0.7.1

03 Jun 2013

- Fix `NameError` ([Issue #3](#)) again - broken in bad merge.

## 7.357 0.7

02 Jun 2013

- Merged Setuptools and Distribute. See `docs/merge.txt` for details.

Added several features that were slated for setuptools 0.6c12:

- Index URL now defaults to HTTPS.
- Added experimental environment marker support. Now clients may designate a [PEP-426](#) environment marker for “extra” dependencies. Setuptools uses this feature in `setup.py` for optional SSL and certificate validation support on older platforms. Based on Distutils-SIG discussions, the syntax is somewhat tentative. There should probably be a PEP with a firmer spec before the feature should be considered suitable for use.
- Added support for SSL certificate validation when installing packages from an HTTPS service.

## 7.358 0.7b4

- [Issue #3](#): Fixed `NameError` in SSL support.

## 7.359 0.6.49

04 Jul 2013

- Move warning check in `get_cache_path` to follow the directory creation to avoid errors when the cache path does not yet exist. Fixes the error reported in [Distribute #375](#).

## 7.360 0.6.48

02 Jul 2013

- Correct `AttributeError` in `ResourceManager.get_cache_path` introduced in 0.6.46 (redo).

## 7.361 0.6.47

02 Jul 2013

- Correct `AttributeError` in `ResourceManager.get_cache_path` introduced in 0.6.46.

## 7.362 0.6.46

29 Jun 2013

- [Distribute #375](#): Issue a warning if the `PYTHON_EGG_CACHE` or otherwise customized egg cache location specifies a directory that's group- or world-writable.

## 7.363 0.6.45

29 May 2013

- [Distribute #379](#): `distribute_setup.py` now traps `VersionConflict` as well, restoring ability to upgrade from an older setuptools version.

## 7.364 0.6.44

28 May 2013

- `distribute_setup.py` has been updated to allow Setuptools 0.7 to satisfy `use_setuptools`.

## 7.365 0.6.43

24 May 2013

- [Distribute #378](#): Restore support for Python 2.4 Syntax (regression in 0.6.42).

## 7.366 0.6.42

24 May 2013

- External links finder no longer yields duplicate links.
- [Distribute #337](#): Moved `site.py` to `setuptools/site-patch.py` (graft of very old patch from setuptools trunk which inspired PR [#31](#)).

## 7.367 0.6.41

24 May 2013

- [Distribute #27](#): Use public api for loading resources from zip files rather than the private method `_zip_directory_cache`.
- Added a new function `easy_install.get_win_launcher` which may be used by third-party libraries such as buildout to get a suitable script launcher.

## 7.368 0.6.40

14 May 2013

- [Distribute #376](#): brought back cli.exe and gui.exe that were deleted in the previous release.

## 7.369 0.6.39

12 May 2013

- Add support for console launchers on ARM platforms.
- Fix possible issue in GUI launchers where the subsystem was not supplied to the linker.
- Launcher build script now refactored for robustness.
- [Distribute #375](#): Resources extracted from a zip egg to the file system now also check the contents of the file against the zip contents during each invocation of `get_resource_filename`.

## 7.370 0.6.38

05 May 2013

- [Distribute #371](#): The launcher manifest file is now installed properly.

## 7.371 0.6.37

04 May 2013

- [Distribute #143](#): Launcher scripts, including `easy_install` itself, are now accompanied by a manifest on 32-bit Windows environments to avoid the Installer Detection Technology and thus undesirable UAC elevation described in [this Microsoft article](#).

## 7.372 0.6.36

05 Apr 2013

- [BB Pull Request #35](#): In [Buildout #64](#), it was reported that under Python 3, installation of `distutils` scripts could attempt to copy the `__pycache__` directory as a file, causing an error, apparently only under Windows. `Easy_install` now skips all directories when processing metadata scripts.

## 7.373 0.6.35

16 Feb 2013

Note this release is backward-incompatible with `distribute` 0.6.23-0.6.34 in how it parses version numbers.

- [Distribute #278](#): Restored compatibility with `distribute` 0.6.22 and `setuptools` 0.6. Updated the documentation to match more closely with the version parsing as intended in `setuptools` 0.6.

## 7.374 0.6.34

30 Dec 2012

- [Distribute #341](#): 0.6.33 fails to build under Python 2.4.

## 7.375 0.6.33

29 Dec 2012

- Fix 2 errors with Jython 2.5.
- Fix 1 failure with Jython 2.5 and 2.7.
- Disable workaround for Jython scripts on Linux systems.
- [Distribute #336](#): *setup.py* no longer masks failure exit code when tests fail.
- Fix issue in *pkg\_resources* where try/except around a platform-dependent import would trigger hook load failures on Mercurial. See pull request 32 for details.
- [Distribute #341](#): Fix a ResourceWarning.

## 7.376 0.6.32

26 Nov 2012

- Fix test suite with Python 2.6.
- Fix some DeprecationWarnings and ResourceWarnings.
- [Distribute #335](#): Backed out *setup\_requires* superceding installed requirements until regression can be addressed.

## 7.377 0.6.31

24 Nov 2012

- [Distribute #303](#): Make sure the manifest only ever contains UTF-8 in Python 3.
- [Distribute #329](#): Properly close files created by tests for compatibility with Jython.
- Work around [Jython #1980](#) and [Jython #1981](#).
- [Distribute #334](#): Provide workaround for packages that reference *sys.\_\_stdout\_\_* such as *numpy* does. This change should address [virtualenv #359](#) <<https://github.com/pypa/virtualenv/issues/359>> ‘\_’ as long as the system encoding is UTF-8 or the IO encoding is specified in the environment, i.e.:

```
PYTHONIOENCODING=utf8 pip install numpy
```

- Fix for encoding issue when installing from Windows executable on Python 3.
- [Distribute #323](#): Allow *setup\_requires* requirements to supercede installed requirements. Added some new keyword arguments to existing *pkg\_resources* methods. Also had to updated how *\_\_path\_\_* is handled for namespace packages to ensure that when a new egg distribution containing a namespace package is placed on

`sys.path`, the entries in `__path__` are found in the same order they would have been in had that egg been on the path when `pkg_resources` was first imported.

## 7.378 0.6.30

22 Oct 2012

- [Distribute #328](#): Clean up temporary directories in `distribute_setup.py`.
- Fix fatal bug in `distribute_setup.py`.

## 7.379 0.6.29

21 Oct 2012

- [BB Pull Request #14](#): Honor file permissions in zip files.
- [Distribute #327](#): Merged pull request [#24](#) to fix a dependency problem with pip.
- Merged pull request [#23](#) to fix <https://github.com/pypa/virtualenv/issues/301>.
- If Sphinx is installed, the `upload_docs` command now runs `build_sphinx` to produce uploadable documentation.
- [Distribute #326](#): `upload_docs` provided mangled auth credentials under Python 3.
- [Distribute #320](#): Fix check for “createable” in `distribute_setup.py`.
- [Distribute #305](#): Remove a warning that was triggered during normal operations.
- [Distribute #311](#): Print metadata in UTF-8 independent of platform.
- [Distribute #303](#): Read manifest file with UTF-8 encoding under Python 3.
- [Distribute #301](#): Allow to run tests of namespace packages when using 2to3.
- [Distribute #304](#): Prevent import loop in `site.py` under Python 3.3.
- [Distribute #283](#): Reenable scanning of `*.pyc` / `*.pyo` files on Python 3.3.
- [Distribute #299](#): The `develop` command didn’t work on Python 3, when using 2to3, as the egg link would go to the Python 2 source. Linking to the 2to3’d code in `build/lib` makes it work, although you will have to rebuild the module before testing it.
- [Distribute #306](#): Even if 2to3 is used, we build in-place under Python 2.
- [Distribute #307](#): Prints the full path when `.svn/entries` is broken.
- [Distribute #313](#): Support for `sdist` subcommands (Python 2.7)
- [Distribute #314](#): `test_local_index()` would fail on OS X.
- [Distribute #310](#): Non-ascii characters in a namespace `__init__.py` causes errors.
- [Distribute #218](#): Improved documentation on behavior of `package_data` and `include_package_data`. Files indicated by `package_data` are now included in the manifest.
- `distribute_setup.py` now allows a `-download-base` argument for retrieving distribute from a specified location.

## 7.380 0.6.28

22 Jul 2012

- [Distribute #294](#): setup.py can now be invoked from any directory.
- Scripts are now installed honoring the umask.
- Added support for .dist-info directories.
- [Distribute #283](#): Fix and disable scanning of \*.pyc / \*.pyo files on Python 3.3.

## 7.381 0.6.27

18 May 2012

- Support current snapshots of CPython 3.3.
- Distribute now recognizes README.rst as a standard, default readme file.
- Exclude 'encodings' modules when removing modules from sys.modules. Workaround for [#285](#).
- [Distribute #231](#): Don't fiddle with system python when used with buildout (bootstrap.py)

## 7.382 0.6.26

08 Apr 2012

- [Distribute #183](#): Symlinked files are now extracted from source distributions.
- [Distribute #227](#): Easy\_install fetch parameters are now passed during the installation of a source distribution; now fulfillment of setup\_requires dependencies will honor the parameters passed to easy\_install.

## 7.383 0.6.25

08 Feb 2012

- [Distribute #258](#): Workaround a cache issue
- [Distribute #260](#): distribute\_setup.py now accepts the --user parameter for Python 2.6 and later.
- [Distribute #262](#): package\_index.open\_with\_auth no longer throws LookupError on Python 3.
- [Distribute #269](#): AttributeError when an exception occurs reading Manifest.in on late releases of Python.
- [Distribute #272](#): Prevent TypeError when namespace package names are unicode and single-install-externally-managed is used. Also fixes PIP issue 449.
- [Distribute #273](#): Legacy script launchers now install with Python2/3 support.

## 7.384 0.6.24

14 Oct 2011

- [Distribute #249](#): Added options to exclude 2to3 fixers

## 7.385 0.6.23

22 Sep 2011

- [Distribute #244](#): Fixed a test
- [Distribute #243](#): Fixed a test
- [Distribute #239](#): Fixed a test
- [Distribute #240](#): Fixed a test
- [Distribute #241](#): Fixed a test
- [Distribute #237](#): Fixed a test
- [Distribute #238](#): easy\_install now uses 64bit executable wrappers on 64bit Python
- [Distribute #208](#): Fixed parsed\_versions, it now honors post-releases as noted in the documentation
- [Distribute #207](#): Windows cli and gui wrappers pass CTRL-C to child python process
- [Distribute #227](#): easy\_install now passes its arguments to setup.py bdist\_egg
- [Distribute #225](#): Fixed a NameError on Python 2.5, 2.4

## 7.386 0.6.21

20 Aug 2011

- [Distribute #225](#): Fixed a regression on py2.4

## 7.387 0.6.20

18 Aug 2011

- [Distribute #135](#): Include url in warning when processing URLs in package\_index.
- [Distribute #212](#): Fix issue where easy\_install fails on Python 3 on windows installer.
- [Distribute #213](#): Fix typo in documentation.

## 7.388 0.6.19

02 Jun 2011

- [Distribute #206](#): AttributeError: 'HTTPMessage' object has no attribute 'getheaders'

## 7.389 0.6.18

01 Jun 2011

- [Distribute #210](#): Fixed a regression introduced by [Distribute #204](#) fix.

## 7.390 0.6.17

30 May 2011

- Support ‘DISTRIBUTE\_DISABLE\_VERSIONED\_EASY\_INSTALL\_SCRIPT’ environment variable to allow to disable installation of easy\_install-\${version} script.
- Support Python >=3.1.4 and >=3.2.1.
- [Distribute #204](#): Don’t try to import the parent of a namespace package in declare\_namespace
- [Distribute #196](#): Tolerate responses with multiple Content-Length headers
- [Distribute #205](#): Sandboxing doesn’t preserve working\_set. Leads to setup\_requires problems.

## 7.391 0.6.16

28 Apr 2011

- Builds sdist gztar even on Windows (avoiding [Distribute #193](#)).
- [Distribute #192](#): Fixed metadata omitted on Windows when package\_dir specified with forward-slash.
- [Distribute #195](#): Cython build support.
- [Distribute #200](#): Issues with recognizing 64-bit packages on Windows.

## 7.392 0.6.15

12 Mar 2011

- Fixed typo in bdist\_egg
- Several issues under Python 3 has been solved.
- [Distribute #146](#): Fixed missing DLL files after easy\_install of windows exe package.

## 7.393 0.6.14

15 Jul 2010

- [Distribute #170](#): Fixed unittest failure. Thanks to Toshio.
- [Distribute #171](#): Fixed race condition in unittests cause deadlocks in test suite.
- [Distribute #143](#): Fixed a lookup issue with easy\_install. Thanks to David and Zooko.
- [Distribute #174](#): Fixed the edit mode when its used with setuptools itself

## 7.394 0.6.13

31 May 2010

- [Distribute #160](#): 2.7 gives ValueError(“Invalid IPv6 URL”)
- [Distribute #150](#): Fixed using ~/.local even in a –no-site-packages virtualenv



- [Distribute #163](#): scan index links before external links, and don't use the md5 when comparing two distributions

## 7.395 0.6.12

06 May 2010

- [Distribute #149](#): Fixed various failures on 2.3/2.4

## 7.396 0.6.11

06 May 2010

- Found another case of SandboxViolation - fixed
- [Distribute #15](#) and [Distribute #48](#): Introduced a socket timeout of 15 seconds on url openings
- Added indexsidebar.html into MANIFEST.in
- [Distribute #108](#): Fixed TypeError with Python3.1
- [Distribute #121](#): Fixed --help install command trying to actually install.
- [Distribute #112](#): Added an os.makedirs so that Tarek's solution will work.
- [Distribute #133](#): Added --no-find-links to easy\_install
- Added easy\_install --user
- [Distribute #100](#): Fixed develop --user not taking '.' in PYTHONPATH into account
- [Distribute #134](#): removed spurious UserWarnings. Patch by VanLindberg
- [Distribute #138](#): cant\_write\_to\_target error when setup\_requires is used.
- [Distribute #147](#): respect the sys.dont\_write\_bytecode flag

## 7.397 0.6.10

12 Dec 2009

- Reverted change made for the DistributionNotFound exception because zc.buildout uses the exception message to get the name of the distribution.

## 7.398 0.6.9

12 Dec 2009

- [Distribute #90](#): unknown setuptools version can be added in the working set
- [Distribute #87](#): setup.py doesn't try to convert distribute\_setup.py anymore Initial Patch by arfrever.
- [Distribute #89](#): added a side bar with a download link to the doc.
- [Distribute #86](#): fixed missing sentence in pkg\_resources doc.
- Added a nicer error message when a DistributionNotFound is raised.

- [Distribute #80](#): test\_develop now works with Python 3.1
- [Distribute #93](#): upload\_docs now works if there is an empty sub-directory.
- [Distribute #70](#): exec bit on non-exec files
- [Distribute #99](#): now the standalone easy\_install command doesn't use a "setup.cfg" if any exists in the working directory. It will use it only if triggered by install\_requires from a setup.py call (install, develop, etc).
- [Distribute #101](#): Allowing os.devnull in Sandbox
- [Distribute #92](#): Fixed the "no eggs" found error with MacPort (platform.mac\_ver() fails)
- [Distribute #103](#): test\_get\_script\_header\_jython\_workaround not run anymore under py3 with C or POSIX local. Contributed by Arfrever.
- [Distribute #104](#): removed the assertion when the installation fails, with a nicer message for the end user.
- [Distribute #100](#): making sure there's no SandboxViolation when the setup script patches setuptools.

## 7.399 0.6.8

01 Nov 2009

- Added "check\_packages" in dist. (added in Setuptools 0.6c11)
- Fixed the DONT\_PATCH\_SETUPTOOLS state.

## 7.400 0.6.7

01 Nov 2009

- [Distribute #58](#): Added --user support to the develop command
- [Distribute #11](#): Generated scripts now wrap their call to the script entry point in the standard "if name == 'main'"
- Added the 'DONT\_PATCH\_SETUPTOOLS' environment variable, so virtualenv can drive an installation that doesn't patch a global setuptools.
- Reviewed unladen-swallow specific change from <http://code.google.com/p/unladen-swallow/source/detail?spec=svn875&r=719> and determined that it no longer applies. Distribute should work fine with Unladen Swallow 2009Q3.
- [Distribute #21](#): Allow PackageIndex.open\_url to gracefully handle all cases of a httplib.HTTPException instead of just InvalidURL and BadStatusLine.
- Removed virtual-python.py from this distribution and updated documentation to point to the actively maintained virtualenv instead.
- [Distribute #64](#): use\_setuptools no longer rebuilds the distribute egg every time it is run
- use\_setuptools now properly respects the requested version
- use\_setuptools will no longer try to import a distribute egg for the wrong Python version
- [Distribute #74](#): no\_fake should be True by default.
- [Distribute #72](#): avoid a bootstrapping issue with easy\_install -U

## 7.401 0.6.6

15 Oct 2009

- Unified the bootstrap file so it works on both py2.x and py3k without 2to3 (patch by Holger Krekel)

## 7.402 0.6.5

15 Oct 2009

- [Distribute #65](#): cli.exe and gui.exe are now generated at build time, depending on the platform in use.
- [Distribute #67](#): Fixed doc typo (PEP 381/PEP 382).
- Distribute no longer shadows setuptools if we require a 0.7-series setuptools. And an error is raised when installing a 0.7 setuptools with distribute.
- When run from within buildout, no attempt is made to modify an existing setuptools egg, whether in a shared egg directory or a system setuptools.
- Fixed a hole in sandboxing allowing builtin file to write outside of the sandbox.

## 7.403 0.6.4

10 Oct 2009

- Added the generation of *distribute\_setup\_3k.py* during the release. This closes [Distribute #52](#).
- Added an upload\_docs command to easily upload project documentation to PyPI's <https://pythonhosted.org>. This close issue [Distribute #56](#).
- Fixed a bootstrap bug on the use\_setuptools() API.

## 7.404 0.6.3

27 Sep 2009

### 7.404.1 setuptools

- Fixed a bunch of calls to file() that caused crashes on Python 3.

### 7.404.2 bootstrapping

- Fixed a bug in sorting that caused bootstrap to fail on Python 3.

## 7.405 0.6.2

26 Sep 2009

### 7.405.1 setuptools

- Added Python 3 support; see docs/python3.txt. This closes [Old Setuptools #39](#).
- Added option to run 2to3 automatically when installing on Python 3. This closes issue [Distribute #31](#).
- Fixed invalid usage of requirement.parse, that broke develop -d. This closes [Old Setuptools #44](#).
- Fixed script launcher for 64-bit Windows. This closes [Old Setuptools #2](#).
- KeyError when compiling extensions. This closes [Old Setuptools #41](#).

### 7.405.2 bootstrapping

- Fixed bootstrap not working on Windows. This closes issue [Distribute #49](#).
- Fixed 2.6 dependencies. This closes issue [Distribute #50](#).
- Make sure setuptools is patched when running through easy\_install This closes [Old Setuptools #40](#).

## 7.406 0.6.1

08 Sep 2009

### 7.406.1 setuptools

- package\_index.urlopen now catches BadStatusLine and malformed url errors. This closes [Distribute #16](#) and [Distribute #18](#).
- zip\_ok is now False by default. This closes [Old Setuptools #33](#).
- Fixed invalid URL error catching. [Old Setuptools #20](#).
- Fixed invalid bootstrapping with easy\_install installation ([Distribute #40](#)). Thanks to Florian Schulze for the help.
- Removed buildout/bootstrap.py. A new repository will create a specific bootstrap.py script.

### 7.406.2 bootstrapping

- The bootstrap process leave setuptools alone if detected in the system and --root or --prefix is provided, but is not in the same location. This closes [Distribute #10](#).

## 7.407 0.6

09 Aug 2009

### 7.407.1 setuptools

- Packages required at build time where not fully present at install time. This closes [Distribute #12](#).
- Protected against failures in tarfile extraction. This closes [Distribute #10](#).
- Made Jython api\_tests.txt doctest compatible. This closes [Distribute #7](#).

- `sandbox.py` replaced builtin type `file` with builtin function `open`. This closes [Distribute #6](#).
- Immediately close all file handles. This closes [Distribute #3](#).
- Added compatibility with Subversion 1.6. This references [Distribute #1](#).

## 7.407.2 pkg\_resources

- Avoid a call to `/usr/bin/sw_vers` on OSX and use the official platform API instead. Based on a patch from ronaldoussoren. This closes issue [#5](#).
- Fixed a `SandboxViolation` for `mkdir` that could occur in certain cases. This closes [Distribute #13](#).
- Allow to find `_on_path` on systems with tight permissions to fail gracefully. This closes [Distribute #9](#).
- Corrected inconsistency between documentation and code of `add_entry`. This closes [Distribute #8](#).
- Immediately close all file handles. This closes [Distribute #3](#).

## 7.407.3 easy\_install

- Immediately close all file handles. This closes [Distribute #3](#).

## 7.408 0.6c9

- Fixed a missing files problem when using Windows source distributions on non-Windows platforms, due to `distutils` not handling manifest file line endings correctly.
- Updated Pyrex support to work with Pyrex 0.9.6 and higher.
- Minor changes for Jython compatibility, including skipping tests that can't work on Jython.
- Fixed not installing eggs in `install_requires` if they were also used for `setup_requires` or `tests_require`.
- Fixed not fetching eggs in `install_requires` when running tests.
- Allow `ez_setup.use_setuptools()` to upgrade existing setuptools installations when called from a standalone `setup.py`.
- Added a warning if a namespace package is declared, but its parent package is not also declared as a namespace.
- Support Subversion 1.5
- Removed use of deprecated `md5` module if `hashlib` is available
- Fixed `bdist_wininst` upload trying to upload the `.exe` twice
- Fixed `bdist_egg` putting a `native_libs.txt` in the source package's `.egg-info`, when it should only be in the built egg's `EGG-INFO`.
- Ensure that `_full_name` is set on all shared libs before extensions are checked for shared lib usage. (Fixes a bug in the experimental shared library build support.)
- Fix to allow unpacked eggs containing native libraries to fail more gracefully under Google App Engine (with an `ImportError` loading the C-based module, instead of getting a `NameError`).
- Fixed `win32.exe` support for `.pth` files, so unnecessary directory nesting is flattened out in the resulting egg. (There was a case-sensitivity problem that affected some distributions, notably `pywin32`.)

- Prevent `--help-commands` and other junk from showing under Python 2.5 when running `easy_install --help`.
- Fixed GUI scripts sometimes not executing on Windows
- Fixed not picking up dependency links from recursive dependencies.
- Only make `.py`, `.dll` and `.so` files executable when unpacking eggs
- Changes for Jython compatibility
- Improved error message when a requirement is also a directory name, but the specified directory is not a source package.
- Fixed `--allow-hosts` option blocking `file:` URLs
- Fixed HTTP SVN detection failing when the page title included a project name (e.g. on SourceForge-hosted SVN)
- Fix Jython script installation to handle `#!` lines better when `sys.executable` is a script.
- Removed use of deprecated `md5` module if `hashlib` is available
- Keep site directories (e.g. `site-packages`) from being included in `.pth` files.

## 7.409 0.6c7

- Fixed `distutils.filelist.findall()` crashing on broken symlinks, and `egg_info` command failing on new, uncommitted SVN directories.
- Fix `import` problems with nested namespace packages installed via `--root` or `--single-version-externally-managed`, due to the parent package not having the child package as an attribute.
- `ftp:` download URLs now work correctly.
- The default `--index-url` is now `https://pypi.python.org/simple`, to use the Python Package Index's new simpler (and faster!) REST API.

## 7.410 0.6c6

- Added `--egg-path` option to `develop` command, allowing you to force `.egg-link` files to use relative paths (allowing them to be shared across platforms on a networked drive).
- Fix not building binary RPMs correctly.
- Fix “eggsecutables” (such as setuptools’ own egg) only being runnable with bash-compatible shells.
- Fix `#!` parsing problems in Windows `.exe` script wrappers, when there was whitespace inside a quoted argument or at the end of the `#!` line (a regression introduced in 0.6c4).
- Fix `test` command possibly failing if an older version of the project being tested was installed on `sys.path` ahead of the test source directory.
- Fix `find_packages()` treating `ez_setup` and directories with `.` in their names as packages.
- EasyInstall no longer aborts the installation process if a URL it wants to retrieve can’t be downloaded, unless the URL is an actual package download. Instead, it issues a warning and tries to keep going.
- Fixed `distutils`-style scripts originally built on Windows having their line endings doubled when installed on any platform.

- Added `--local-snapshots-ok` flag, to allow building eggs from projects installed using `setup.py develop`.
- Fixed not HTML-decoding URLs scraped from web pages

## 7.411 0.6c5

- Fix uploaded `bdist_rpm` packages being described as `bdist_egg` packages under Python versions less than 2.5.
- Fix uploaded `bdist_wininst` packages being described as suitable for “any” version by Python 2.5, even if a `--target-version` was specified.
- Fixed `.dll` files on Cygwin not having executable permissions when an egg is installed unzipped.

## 7.412 0.6c4

- Overhauled Windows script wrapping to support `bdist_wininst` better. Scripts installed with `bdist_wininst` will always use `#!python.exe` or `#!pythonw.exe` as the executable name (even when built on non-Windows platforms!), and the wrappers will look for the executable in the script’s parent directory (which should find the right version of Python).
- Fix upload command not uploading files built by `bdist_rpm` or `bdist_wininst` under Python 2.3 and 2.4.
- Add support for “eggsecutable” headers: a `#!/bin/sh` script that is prepended to an `.egg` file to allow it to be run as a script on Unix-ish platforms. (This is mainly so that setuptools itself can have a single-file installer on Unix, without doing multiple downloads, dealing with firewalls, etc.)
- Fix problem with empty revision numbers in Subversion 1.4 `entries` files
- Use cross-platform relative paths in `easy-install.pth` when doing `develop` and the source directory is a subdirectory of the installation target directory.
- Fix a problem installing eggs with a system packaging tool if the project contained an implicit namespace package; for example if the `setup()` listed a namespace package `foo.bar` without explicitly listing `foo` as a namespace package.
- Added support for HTTP “Basic” authentication using `http://user:pass@host` URLs. If a password-protected page contains links to the same host (and protocol), those links will inherit the credentials used to access the original page.
- Removed all special support for Sourceforge mirrors, as Sourceforge’s mirror system now works well for non-browser downloads.
- Fixed not recognizing `win32.exe` installers that included a custom bitmap.
- Fixed not allowing `os.open()` of paths outside the sandbox, even if they are opened read-only (e.g. reading `/dev/urandom` for random numbers, as is done by `os.urandom()` on some platforms).
- Fixed a problem with `.pth` testing on Windows when `sys.executable` has a space in it (e.g., the user installed Python to a Program Files directory).

## 7.413 0.6c3

- Fixed breakages caused by Subversion 1.4’s new “working copy” format

- You can once again use “python -m easy\_install” with Python 2.4 and above.
- Python 2.5 compatibility fixes added.

## 7.414 0.6c2

- The `ez_setup` module displays the conflicting version of setuptools (and its installation location) when a script requests a version that’s not available.
- Running `setup.py develop` on a setuptools-using project will now install setuptools if needed, instead of only downloading the egg.
- Windows script wrappers now support quoted arguments and arguments containing spaces. (Patch contributed by Jim Fulton.)
- The `ez_setup.py` script now actually works when you put a setuptools .egg alongside it for bootstrapping an offline machine.
- A writable installation directory on `sys.path` is no longer required to download and extract a source distribution using `--editable`.
- Generated scripts now use `-x` on the `#!` line when `sys.executable` contains non-ASCII characters, to prevent deprecation warnings about an unspecified encoding when the script is run.

## 7.415 0.6c1

- Fixed `AttributeError` when trying to download a `setup_requires` dependency when a distribution lacks a `dependency_links` setting.
- Made `zip-safe` and `not-zip-safe` flag files contain a single byte, so as to play better with packaging tools that complain about zero-length files.
- Made `setup.py develop` respect the `--no-deps` option, which it previously was ignoring.
- Support `extra_path` option to `setup()` when `install` is run in backward-compatibility mode.
- Source distributions now always include a `setup.cfg` file that explicitly sets `egg_info` options such that they produce an identical version number to the source distribution’s version number. (Previously, the default version number could be different due to the use of `--tag-date`, or if the version was overridden on the command line that built the source distribution.)
- `EasyInstall` now includes setuptools version information in the `User-Agent` string sent to websites it visits.

## 7.416 0.6b4

- Fix `register` not obeying name/version set by `egg_info` command, if `egg_info` wasn’t explicitly run first on the same command line.
- Added `--no-date` and `--no-svn-revision` options to `egg_info` command, to allow suppressing tags configured in `setup.cfg`.
- Fixed redundant warnings about missing `README` file(s); it should now appear only if you are actually a source distribution.
- Fix creating Python wrappers for non-Python scripts



- Fix `ftp://` directory listing URLs from causing a crash when used in the “Home page” or “Download URL” slots on PyPI.
- Fix `sys.path_importer_cache` not being updated when an existing zipfile or directory is deleted/overwritten.
- Fix not recognizing HTML 404 pages from package indexes.
- Allow `file://` URLs to be used as a package index. URLs that refer to directories will use an internally-generated directory listing if there is no `index.html` file in the directory.
- Allow external links in a package index to be specified using `rel="homepage"` or `rel="download"`, without needing the old PyPI-specific visible markup.
- Suppressed warning message about possibly-misspelled project name, if an egg or link for that project name has already been seen.

## 7.417 0.6b3

- Fix `bdist_egg` not including files in subdirectories of `.egg-info`.
- Allow `.py` files found by the `include_package_data` option to be automatically included. Remove duplicate data file matches if both `include_package_data` and `package_data` are used to refer to the same files.
- Fix local `--find-links` eggs not being copied except with `--always-copy`.
- Fix sometimes not detecting local packages installed outside of “site” directories.
- Fix mysterious errors during initial `setuptools` install, caused by `ez_setup` trying to run `easy_install` twice, due to a code fallthru after deleting the egg from which it’s running.

## 7.418 0.6b2

- Don’t install or update a `site.py` patch when installing to a `PYTHONPATH` directory with `--multi-version`, unless an `easy-install.pth` file is already in use there.
- Construct `.pth` file paths in such a way that installing an egg whose name begins with `import` doesn’t cause a syntax error.
- Fixed a bogus warning message that wasn’t updated since the 0.5 versions.

## 7.419 0.6b1

- Strip `module` from the end of compiled extension modules when computing the name of a `.py` loader/wrapper. (Python’s import machinery ignores this suffix when searching for an extension module.)
- Better ambiguity management: accept `#egg` name/version even if processing what appears to be a correctly-named distutils file, and ignore `.egg` files with no `-`, since valid Python `.egg` files always have a version number (but Scheme eggs often don’t).
- Support `file://` links to directories in `--find-links`, so that `easy_install` can build packages from local source checkouts.

- Added automatic retry for Sourceforge mirrors. The new download process is to first just try `dl.sourceforge.net`, then randomly select mirror IPs and remove ones that fail, until something works. The removed IPs stay removed for the remainder of the run.
- Ignore `bdist_dumb` distributions when looking at download URLs.

## 7.420 0.6a11

- Added `test_loader` keyword to support custom test loaders
- Added `setuptools.file_finders` entry point group to allow implementing revision control plugins.
- Added `--identity` option to upload command.
- Added `dependency_links` to allow specifying URLs for `--find-links`.
- Enhanced test loader to scan packages as well as modules, and call `additional_tests()` if present to get non-unittest tests.
- Support namespace packages in conjunction with system packagers, by omitting the installation of any `__init__.py` files for namespace packages, and adding a special `.pth` file to create a working package in `sys.modules`.
- Made `--single-version-externally-managed` automatic when `--root` is used, so that most system packagers won't require special support for setuptools.
- Fixed `setup_requires`, `tests_require`, etc. not using `setup.cfg` or other configuration files for their option defaults when installing, and also made the install use `--multi-version` mode so that the project directory doesn't need to support `.pth` files.
- `MANIFEST.in` is now forcibly closed when any errors occur while reading it. Previously, the file could be left open and the actual error would be masked by problems trying to remove the open file on Windows systems.
- Process `dependency_links.txt` if found in a distribution, by adding the URLs to the list for scanning.
- Use relative paths in `.pth` files when eggs are being installed to the same directory as the `.pth` file. This maximizes portability of the target directory when building applications that contain eggs.
- Added `easy_install-N.N` script(s) for convenience when using multiple Python versions.
- Added automatic handling of installation conflicts. Eggs are now shifted to the front of `sys.path`, in an order consistent with where they came from, making EasyInstall seamlessly co-operate with system package managers.

The `--delete-conflicting` and `--ignore-conflicts-at-my-risk` options are now no longer necessary, and will generate warnings at the end of a run if you use them.

- Don't recursively traverse subdirectories given to `--find-links`.

## 7.421 0.6a10

- Fixed the `develop` command ignoring `--find-links`.
- Added exhaustive testing of the install directory, including a spawn test for `.pth` file support, and directory writability/existence checks. This should virtually eliminate the need to set or configure `--site-dirs`.
- Added `--prefix` option for more do-what-I-mean-ishness in the absence of RTFM-ing. :)
- Enhanced `PYTHONPATH` support so that you don't have to put any eggs on it manually to make it work. `--multi-version` is no longer a silent default; you must explicitly use it if installing to a non-`PYTHONPATH`, non-`"site"` directory.

- Expand \$variables used in the `--site-dirs`, `--build-directory`, `--install-dir`, and `--script-dir` options, whether on the command line or in configuration files.
- Improved SourceForge mirror processing to work faster and be less affected by transient HTML changes made by SourceForge.
- PyPI searches now use the exact spelling of requirements specified on the command line or in a project's `install_requires`. Previously, a normalized form of the name was used, which could lead to unnecessary full-index searches when a project's name had an underscore (`_`) in it.
- EasyInstall can now download bare `.py` files and wrap them in an egg, as long as you include an `#egg=name-version` suffix on the URL, or if the `.py` file is listed as the “Download URL” on the project's PyPI page. This allows third parties to “package” trivial Python modules just by linking to them (e.g. from within their own PyPI page or download links page).
- The `--always-copy` option now skips “system” and “development” eggs since they can't be reliably copied. Note that this may cause EasyInstall to choose an older version of a package than what you expected, or it may cause downloading and installation of a fresh version of what's already installed.
- The `--find-links` option previously scanned all supplied URLs and directories as early as possible, but now only directories and direct archive links are scanned immediately. URLs are not retrieved unless a package search was already going to go online due to a package not being available locally, or due to the use of the `--update` or `-U` option.
- Fixed the annoying `--help-commands` wart.

## 7.422 0.6a9

- The `sdist` command no longer uses the traditional `MANIFEST` file to create source distributions. `MANIFEST.in` is still read and processed, as are the standard defaults and pruning. But the manifest is built inside the project's `.egg-info` directory as `SOURCES.txt`, and it is rebuilt every time the `egg_info` command is run.
- Added the `include_package_data` keyword to `setup()`, allowing you to automatically include any package data listed in revision control or `MANIFEST.in`
- Added the `exclude_package_data` keyword to `setup()`, allowing you to trim back files included via the `package_data` and `include_package_data` options.
- Fixed `--tag-svn-revision` not working when run from a source distribution.
- Added warning for namespace packages with missing `declare_namespace()`
- Added `tests_require` keyword to `setup()`, so that e.g. packages requiring `nose` to run unit tests can make this dependency optional unless the `test` command is run.
- Made all commands that use `easy_install` respect its configuration options, as this was causing some problems with `setup.py install`.
- Added an `unpack_directory()` driver to `setuptools.archive_util`, so that you can process a directory tree through a processing filter as if it were a zipfile or tarfile.
- Added an internal `install_egg_info` command to use as part of old-style `install` operations, that installs an `.egg-info` directory with the package.
- Added a `--single-version-externally-managed` option to the `install` command so that you can more easily wrap a “flat” egg in a system package.
- Enhanced `bdist_rpm` so that it installs single-version eggs that don't rely on a `.pth` file. The `--no-egg` option has been removed, since all RPMs are now built in a more backwards-compatible format.

- Support full roundtrip translation of eggs to and from `bdist_wininst` format. Running `bdist_wininst` on a setuptools-based package wraps the egg in an `.exe` that will safely install it as an egg (i.e., with metadata and entry-point wrapper scripts), and `easy_install` can turn the `.exe` back into an `.egg` file or directory and install it as such.
- Fixed `.pth` file processing picking up nested eggs (i.e. ones inside “baskets”) when they weren’t explicitly listed in the `.pth` file.
- If more than one URL appears to describe the exact same distribution, prefer the shortest one. This helps to avoid “table of contents” CGI URLs like the ones on [effbot.org](http://effbot.org).
- Quote arguments to `python.exe` (including `python`’s path) to avoid problems when Python (or a script) is installed in a directory whose name contains spaces on Windows.
- Support full roundtrip translation of eggs to and from `bdist_wininst` format. Running `bdist_wininst` on a setuptools-based package wraps the egg in an `.exe` that will safely install it as an egg (i.e., with metadata and entry-point wrapper scripts), and `easy_install` can turn the `.exe` back into an `.egg` file or directory and install it as such.

## 7.423 0.6a8

- Fixed some problems building extensions when Pyrex was installed, especially with Python 2.4 and/or packages using SWIG.
- Made `develop` command accept all the same options as `easy_install`, and use the `easy_install` command’s configuration settings as defaults.
- Made `egg_info --tag-svn-revision` fall back to extracting the revision number from `PKG-INFO` in case it is being run on a source distribution of a snapshot taken from a Subversion-based project.
- Automatically detect `.dll`, `.so` and `.dylib` files that are being installed as data, adding them to `native_libs.txt` automatically.
- Fixed some problems with fresh checkouts of projects that don’t include `.egg-info/PKG-INFO` under revision control and put the project’s source code directly in the project directory. If such a package had any requirements that get processed before the `egg_info` command can be run, the setup scripts would fail with a “Missing ‘Version:’ header and/or `PKG-INFO` file” error, because the egg runtime interpreted the unbuilt metadata in a directory on `sys.path` (i.e. the current directory) as being a corrupted egg. Setuptools now monkeypatches the distribution metadata cache to pretend that the egg has valid version information, until it has a chance to make it actually be so (via the `egg_info` command).
- Update for changed SourceForge mirror format
- Fixed not installing dependencies for some packages fetched via Subversion
- Fixed dependency installation with `--always-copy` not using the same dependency resolution procedure as other operations.
- Fixed not fully removing temporary directories on Windows, if a Subversion checkout left read-only files behind
- Fixed some problems building extensions when Pyrex was installed, especially with Python 2.4 and/or packages using SWIG.

## 7.424 0.6a7

- Fixed not being able to install Windows script wrappers using Python 2.3

## 7.425 0.6a6

- Added support for “traditional” PYTHONPATH-based non-root installation, and also the convenient `virtual-python.py` script, based on a contribution by Ian Bicking. The setuptools egg now contains a hacked `site` module that makes the PYTHONPATH-based approach work with `.pth` files, so that you can get the full EasyInstall feature set on such installations.
- Added `--no-deps` and `--allow-hosts` options.
- Improved Windows `.exe` script wrappers so that the script can have the same name as a module without confusing Python.
- Changed dependency processing so that it’s breadth-first, allowing a depender’s preferences to override those of a dependee, to prevent conflicts when a lower version is acceptable to the dependee, but not the depender. Also, ensure that currently installed/selected packages aren’t given precedence over ones desired by a package being installed, which could cause conflict errors.

## 7.426 0.6a5

- Fixed missing `gui/cli.exe` files in distribution. Fixed bugs in tests.

## 7.427 0.6a3

- Added `gui_scripts` entry point group to allow installing GUI scripts on Windows and other platforms. (The special handling is only for Windows; other platforms are treated the same as for `console_scripts`.)
- Improved error message when trying to use old ways of running `easy_install`. Removed the ability to run via `python -m` or by running `easy_install.py`; `easy_install` is the command to run on all supported platforms.
- Improved wrapper script generation and runtime initialization so that a `VersionConflict` doesn’t occur if you later install a competing version of a needed package as the default version of that package.
- Fixed a problem parsing version numbers in `#egg=` links.

## 7.428 0.6a2

- Added `console_scripts` entry point group to allow installing scripts without the need to create separate script files. On Windows, console scripts get an `.exe` wrapper so you can just type their name. On other platforms, the scripts are written without a file extension.
- EasyInstall can now install “console\_scripts” defined by packages that use `setuptools` and define appropriate entry points. On Windows, console scripts get an `.exe` wrapper so you can just type their name. On other platforms, the scripts are installed without a file extension.
- Using `python -m easy_install` or running `easy_install.py` is now **DEPRECATED**, since an `easy_install` wrapper is now available on all platforms.

## 7.429 0.6a1

- Added support for building “old-style” RPMs that don’t install an egg for the target package, using a `--no-egg` option.
- The `build_ext` command now works better when using the `--inplace` option and multiple Python versions. It now makes sure that all extensions match the current Python version, even if newer copies were built for a different Python version.
- The `upload` command no longer attaches an extra `.zip` when uploading eggs, as PyPI now supports egg uploads without trickery.
- The `ez_setup` script/module now displays a warning before downloading the setuptools egg, and attempts to check the downloaded egg against an internal MD5 checksum table.
- Fixed the `--tag-svn-revision` option of `egg_info` not finding the latest revision number; it was using the revision number of the directory containing `setup.py`, not the highest revision number in the project.
- Added `eager_resources` setup argument
- The `sdist` command now recognizes Subversion “deleted file” entries and does not include them in source distributions.
- `setuptools` now embeds itself more thoroughly into the distutils, so that other distutils extensions (e.g. `py2exe`, `py2app`) will subclass setuptools’ versions of things, rather than the native distutils ones.
- Added `entry_points` and `setup_requires` arguments to `setup()`; `setup_requires` allows you to automatically find and download packages that are needed in order to *build* your project (as opposed to running it).
- `setuptools` now finds its commands, `setup()` argument validators, and metadata writers using entry points, so that they can be extended by third-party packages. See [Creating distutils Extensions](#) for more details.
- The vestigial `depends` command has been removed. It was never finished or documented, and never would have worked without EasyInstall - which it pre-dated and was never compatible with.
- EasyInstall now does MD5 validation of downloads from PyPI, or from any link that has an “#md5=...” trailer with a 32-digit lowercase hex md5 digest.
- EasyInstall now handles symlinks in target directories by removing the link, rather than attempting to overwrite the link’s destination. This makes it easier to set up an alternate Python “home” directory (as described in the Non-Root Installation section of the docs).
- Added support for handling MacOS platform information in `.egg` filenames, based on a contribution by Kevin Dangoor. You may wish to delete and reinstall any eggs whose filename includes “darwin” and “Power\_Macintosh”, because the format for this platform information has changed so that minor OS X upgrades (such as 10.4.1 to 10.4.2) do not cause eggs built with a previous OS version to become obsolete.
- `easy_install`’s dependency processing algorithms have changed. When using `--always-copy`, it now ensures that dependencies are copied too. When not using `--always-copy`, it tries to use a single resolution loop, rather than recursing.
- Fixed installing extra `.pyc` or `.pyo` files for scripts with `.py` extensions.
- Added `--site-dirs` option to allow adding custom “site” directories. Made `easy-install.pth` work in platform-specific alternate site directories (e.g. `~/Library/Python/2.x/site-packages` on Mac OS X).
- If you manually delete the current version of a package, the next run of EasyInstall against the target directory will now remove the stray entry from the `easy-install.pth` file.

- EasyInstall now recognizes URLs with a `#egg=project_name` fragment ID as pointing to the named project's source checkout. Such URLs have a lower match precedence than any other kind of distribution, so they'll only be used if they have a higher version number than any other available distribution, or if you use the `--editable` option. The `#egg` fragment can contain a version if it's formatted as `#egg=proj-ver`, where `proj` is the project name, and `ver` is the version number. You *must* use the format for these values that the `bdist_egg` command uses; i.e., all non-alphanumeric runs must be condensed to single underscore characters.
- Added the `--editable` option; see Editing and Viewing Source Packages in the docs. Also, slightly changed the behavior of the `--build-directory` option.
- Fixed the setup script sandbox facility not recognizing certain paths as valid on case-insensitive platforms.

## 7.430 0.5a12

- The zip-safety scanner now checks for modules that might be used with `python -m`, and marks them as unsafe for zipping, since Python 2.4 can't handle `-m` on zipped modules.
- Fix `python -m easy_install` not working due to setuptools being installed as a zipfile. Update safety scanner to check for modules that might be used as `python -m` scripts.
- Misc. fixes for win32.exe support, including changes to support Python 2.4's changed `bdist_wininst` format.

## 7.431 0.5a11

- Fix breakage of the "develop" command that was caused by the addition of `--always-unzip` to the `easy_install` command.

## 7.432 0.5a10

- Put the `easy_install` module back in as a module, as it's needed for `python -m` to run it!
- Allow `--find-links/-f` to accept local directories or filenames as well as URLs.

## 7.433 0.5a9

- Include `svn:externals` directories in source distributions as well as normal subversion-controlled files and directories.
- Added `exclude=patternlist` option to `setuptools.find_packages()`
- Changed `-tag-svn-revision` to include an "r" in front of the revision number for better readability.
- Added ability to build eggs without including source files (except for any scripts, of course), using the `--exclude-source-files` option to `bdist_egg`.
- `setup.py install` now automatically detects when an "unmanaged" package or module is going to be on `sys.path` ahead of a package being installed, thereby preventing the newer version from being imported. If this occurs, a warning message is output to `sys.stderr`, but installation proceeds anyway. The warning message informs the user what files or directories need deleting, and advises them they can also use EasyInstall (with the `--delete-conflicting` option) to do it automatically.



- The `egg_info` command now adds a `top_level.txt` file to the metadata directory that lists all top-level modules and packages in the distribution. This is used by the `easy_install` command to find possibly-conflicting “unmanaged” packages when installing the distribution.
- Added `zip_safe` and `namespace_packages` arguments to `setup()`. Added package analysis to determine zip-safety if the `zip_safe` flag is not given, and advise the author regarding what code might need changing.
- Fixed the swapped `-d` and `-b` options of `bdist_egg`.
- EasyInstall now automatically detects when an “unmanaged” package or module is going to be on `sys.path` ahead of a package you’re installing, thereby preventing the newer version from being imported. By default, it will abort installation to alert you of the problem, but there are also new options (`--delete-conflicting` and `--ignore-conflicts-at-my-risk`) available to change the default behavior. (Note: this new feature doesn’t take effect for egg files that were built with older setuptools versions, because they lack the new metadata file required to implement it.)
- The `easy_install distutils` command now uses `DistutilsError` as its base error type for errors that should just issue a message to `stderr` and exit the program without a traceback.
- EasyInstall can now be given a path to a directory containing a setup script, and it will attempt to build and install the package there.
- EasyInstall now performs a safety analysis on module contents to determine whether a package is likely to run in zipped form, and displays information about what modules may be doing introspection that would break when running as a zipfile.
- Added the `--always-unzip/-Z` option, to force unzipping of packages that would ordinarily be considered safe to unzip, and changed the meaning of `--zip-ok/-z` to “always leave everything zipped”.

## 7.434 0.5a8

- The “`egg_info`” command now always sets the distribution metadata to “safe” forms of the distribution name and version, so that distribution files will be generated with parseable names (i.e., ones that don’t include ‘-’ in the name or version). Also, this means that if you use the various `--tag` options of “`egg_info`”, any distributions generated will use the tags in the version, not just egg distributions.
- Added support for defining command aliases in distutils configuration files, under the “[aliases]” section. To prevent recursion and to allow aliases to call the command of the same name, a given alias can be expanded only once per command-line invocation. You can define new aliases with the “`alias`” command, either for the local, global, or per-user configuration.
- Added “`rotate`” command to delete old distribution files, given a set of patterns to match and the number of files to keep. (Keeps the most recently-modified distribution files matching each pattern.)
- Added “`saveopts`” command that saves all command-line options for the current invocation to the local, global, or per-user configuration file. Useful for setting defaults without having to hand-edit a configuration file.
- Added a “`setopt`” command that sets a single option in a specified distutils configuration file.
- There is now a separate documentation page for setuptools; revision history that’s not specific to EasyInstall has been moved to that page.

## 7.435 0.5a7

- Added “`upload`” support for egg and source distributions, including a bug fix for “`upload`” and a temporary workaround for lack of .egg support in PyPI.



## 7.436 0.5a6

- Beefed up the “sdist” command so that if you don’t have a MANIFEST.in, it will include all files under revision control (CVS or Subversion) in the current directory, and it will regenerate the list every time you create a source distribution, not just when you tell it to. This should make the default “do what you mean” more often than the distutils’ default behavior did, while still retaining the old behavior in the presence of MANIFEST.in.
- Fixed the “develop” command always updating .pth files, even if you specified `-n` or `--dry-run`.
- Slightly changed the format of the generated version when you use `--tag-build` on the “egg\_info” command, so that you can make tagged revisions compare *lower* than the version specified in setup.py (e.g. by using `--tag-build=dev`).

## 7.437 0.5a5

- Added `develop` command to setuptools-based packages. This command installs an `.egg-link` pointing to the package’s source directory, and script wrappers that `execfile()` the source versions of the package’s scripts. This lets you put your development checkout(s) on `sys.path` without having to actually install them. (To uninstall the link, use `setup.py develop --uninstall`.)
- Added `egg_info` command to setuptools-based packages. This command just creates or updates the “projectname.egg-info” directory, without building an egg. (It’s used by the `bdist_egg`, `test`, and `develop` commands.)
- Enhanced the `test` command so that it doesn’t install the package, but instead builds any C extensions in-place, updates the `.egg-info` metadata, adds the source directory to `sys.path`, and runs the tests directly on the source. This avoids an “unmanaged” installation of the package to `site-packages` or elsewhere.
- Made `easy_install` a standard setuptools command, moving it from the `easy_install` module to `setuptools.command.easy_install`. Note that if you were importing or extending it, you must now change your imports accordingly. `easy_install.py` is still installed as a script, but not as a module.

## 7.438 0.5a4

- Setup scripts using setuptools can now list their dependencies directly in the setup.py file, without having to manually create a `depends.txt` file. The `install_requires` and `extras_require` arguments to `setup()` are used to create a dependencies file automatically. If you are manually creating `depends.txt` right now, please switch to using these setup arguments as soon as practical, because `depends.txt` support will be removed in the 0.6 release cycle. For documentation on the new arguments, see the `setuptools.dist.Distribution` class.
- Setup scripts using setuptools now always install using `easy_install` internally, for ease of uninstallation and upgrading.
- Added `--always-copy/-a` option to always copy needed packages to the installation directory, even if they’re already present elsewhere on `sys.path`. (In previous versions, this was the default behavior, but now you must request it.)
- Added `--upgrade/-U` option to force checking PyPI for latest available version(s) of all packages requested by name and version, even if a matching version is available locally.
- Added automatic installation of dependencies declared by a distribution being installed. These dependencies must be listed in the distribution’s `EGG-INFO` directory, so the distribution has to have declared its dependencies

by using setuptools. If a package has requirements it didn't declare, you'll still have to deal with them yourself. (E.g., by asking EasyInstall to find and install them.)

- Added the `--record` option to `easy_install` for the benefit of tools that run `setup.py install --record=filename` on behalf of another packaging system.)

## 7.439 0.5a3

- Fixed not setting script permissions to allow execution.
- Improved sandboxing so that setup scripts that want a temporary directory (e.g. pychecker) can still run in the sandbox.

## 7.440 0.5a2

- Fix stupid stupid refactoring-at-the-last-minute typos. :(

## 7.441 0.5a1

- Added support for “self-installation” bootstrapping. Packages can now include `ez_setup.py` in their source distribution, and add the following to their `setup.py`, in order to automatically bootstrap installation of setuptools as part of their setup process:

```
from ez_setup import use_setuptools
use_setuptools()

from setuptools import setup
# etc...
```

- Added support for converting `.win32.exe` installers to eggs on the fly. EasyInstall will now recognize such files by name and install them.
- Fixed a problem with picking the “best” version to install (versions were being sorted as strings, rather than as parsed values)

## 7.442 0.4a4

- Added support for the distutils “verbose/quiet” and “dry-run” options, as well as the “optimize” flag.
- Support downloading packages that were uploaded to PyPI (by scanning all links on package pages, not just the homepage/download links).

## 7.443 0.4a3

- Add progress messages to the search/download process so that you can tell what URLs it's reading to find download links. (Hopefully, this will help people report out-of-date and broken links to package authors, and to tell when they've asked for a package that doesn't exist.)

## 7.444 0.4a2

- Added `ez_setup.py` installer/bootstrap script to make initial setuptools installation easier, and to allow distributions using setuptools to avoid having to include setuptools in their source distribution.
- All downloads are now managed by the `PackageIndex` class (which is now subclassable and replaceable), so that embedders can more easily override download logic, give download progress reports, etc. The class has also been moved to the new `setuptools.package_index` module.
- The `Installer` class no longer handles downloading, manages a temporary directory, or tracks the `zip_ok` option. Downloading is now handled by `PackageIndex`, and `Installer` has become an `easy_install` command class based on `setuptools.Command`.
- There is a new `setuptools.sandbox.run_setup()` API to invoke a setup script in a directory sandbox, and a new `setuptools.archive_util` module with an `unpack_archive()` API. These were split out of `EasyInstall` to allow reuse by other tools and applications.
- `setuptools.Command` now supports reinitializing commands using keyword arguments to set/reset options. Also, `Command` subclasses can now set their `command_consumes_arguments` attribute to `True` in order to receive an `args` option containing the rest of the command line.
- Added support for installing scripts
- Added support for setting options via distutils configuration files, and using distutils' default options as a basis for `EasyInstall`'s defaults.
- Renamed `--scan-url/-s` to `--find-links/-f` to free up `-s` for the script installation directory option.
- Use `urllib2` instead of `urllib`, to allow use of `https:` URLs if Python includes SSL support.

## 7.445 0.4a1

- Added `--scan-url` and `--index-url` options, to scan download pages and search PyPI for needed packages.

## 7.446 0.3a4

- Restrict `--build-directory=DIR/-b DIR` option to only be used with single URL installs, to avoid running the wrong `setup.py`.

## 7.447 0.3a3

- Added `--build-directory=DIR/-b DIR` option.
- Added “installation report” that explains how to use ‘`require()`’ when doing a multiversion install or alternate installation directory.
- Added SourceForge mirror auto-select (Contributed by Ian Bicking)
- Added “sandboxing” that stops a setup script from running if it attempts to write to the filesystem outside of the build area
- Added more workarounds for packages with quirky `install_data` hacks

## 7.448 0.3a2

- Added new options to `bdist_egg` to allow tagging the egg's version number with a subversion revision number, the current date, or an explicit tag value. Run `setup.py bdist_egg --help` to get more information.
- Added subversion download support for `svn:` and `svn+` URLs, as well as automatic recognition of HTTP subversion URLs (Contributed by Ian Bicking)
- Misc. bug fixes

## 7.449 0.3a1

- Initial release.

## CHAPTER 8

---

### Credits

---

- The original design for the `.egg` format and the `pkg_resources` API was co-created by Phillip Eby and Bob Ippolito. Bob also implemented the first version of `pkg_resources`, and supplied the OS X operating system version compatibility algorithm.
- Ian Bicking implemented many early “creature comfort” features of `easy_install`, including support for downloading via Sourceforge and Subversion repositories. Ian’s comments on the Web-SIG about WSGI application deployment also inspired the concept of “entry points” in eggs, and he has given talks at PyCon and elsewhere to inform and educate the community about eggs and `setuptools`.
- Jim Fulton contributed time and effort to build automated tests of various aspects of `easy_install`, and supplied the doctests for the command-line `.exe` wrappers on Windows.
- Phillip J. Eby is the seminal author of `setuptools`, and first proposed the idea of an importable binary distribution format for Python application plug-ins.
- Significant parts of the implementation of `setuptools` were funded by the Open Source Applications Foundation, to provide a plug-in infrastructure for the Chandler PIM application. In addition, many OSAF staffers (such as Mike “Code Bear” Taylor) contributed their time and stress as guinea pigs for the use of eggs and `setuptools`, even before eggs were “cool”. (Thanks, guys!)
- Tarek Ziadé is the principal author of the `Distribute` fork, which re-invigorated the community on the project, encouraged renewed innovation, and addressed many defects.
- Jason R. Coombs performed the merge with `Distribute`, maintaining the project for several years in coordination with the Python Packaging Authority (PyPA).



## P

Python Enhancement Proposals

PEP 370, [96](#)