



# Bazaar Integration Guide

*Release 2.8.0dev1*

**Bazaar Developers**

March 22, 2019

## Contents

<b>1</b>	<b>Starting with bzrlib</b>	<b>ii</b>
1.1	Within bzzr . . . . .	ii
1.2	From outside bzzr . . . . .	ii
<b>2</b>	<b>Running bzzr commands</b>	<b>ii</b>
<b>3</b>	<b>Manipulating the Working Tree</b>	<b>ii</b>
3.1	Compare trees . . . . .	iii
3.2	Adding Files . . . . .	iii
3.3	Removing Files . . . . .	iii
3.4	Renaming a File . . . . .	iv
3.5	Moving Files . . . . .	iv
3.6	Committing Changes . . . . .	iv
<b>4</b>	<b>Generating a Log for a File</b>	<b>iv</b>
<b>5</b>	<b>Annotating a File</b>	<b>iv</b>
<b>6</b>	<b>Working with branches</b>	<b>v</b>
6.1	Branching from an existing branch . . . . .	v
6.2	Pushing and pulling branches . . . . .	v
<b>7</b>	<b>Checkout from an existing branch</b>	<b>vi</b>
<b>8</b>	<b>History Operations</b>	<b>vi</b>
8.1	Finding the last revision number or id . . . . .	vi
8.2	Getting the list of revision ids that make up a branch . . . . .	vi
8.3	Getting a Revision object from a revision id . . . . .	vi
8.4	Accessing the files from a revision . . . . .	vii

---

This document provides some general observations on integrating with Bazaar and some recipes for typical tasks. It is intended to be useful to someone developing either a plugin or some other piece of software that integrates with bazaar. If you want to know about a topic that's not covered here, just ask us.

## 1 Starting with bzrlib

### 1.1 Within bazaar

When using bzrlib within the bazaar program (for instance as a bazaar plugin), bzrlib's global state is already available for use.

### 1.2 From outside bazaar

To use bzrlib outside of bazaar some global state needs to be setup. bzrlib needs ways to handle user input, passwords, a place to emit progress bars, logging setup appropriately for your program. The easiest way to set all this up in the same fashion bazaar does is to call `bzrlib.initialize`.

This returns a context manager within which bzrlib functions will work correctly. See the pydoc for `bzrlib.initialize` for more information. (You can get away without entering the context manager, because the setup work happens directly from `initialize`.)

In Python 2.4 the `with` keyword is not supported and so you need to use the context manager manually:

```
# This sets up your ~/.bazaar.log, ui factory and so on and so forth. It is  
# not safe to use as a doctest.  
library_state = bzrlib.initialize()  
library_state.__enter__()  
try:  
    pass  
    # do stuff here  
finally:  
    library_state.__exit__(None, None, None)
```

## 2 Running bazaar commands

To run command-line commands in-process:

```
from bzrlib.commands import get_command  
  
cmd = get_command('version')  
cmd.run([])
```

This will send output through the current UIFactory; you can redirect this elsewhere through the parameters to `bzrlib.initialize`.

## 3 Manipulating the Working Tree

Most objects in Bazaar are in files, named after the class they contain. To manipulate the Working Tree we need a valid WorkingTree object, which is loaded from the `workingtree.py` file, eg:

```
from bzrlib import workingtree
wt = workingtree.WorkingTree.open('/home/jebw/bzrtest')
```

This gives us a WorkingTree object, which has various methods spread over itself, and its parent classes MutableTree and Tree - it's worth having a look through these three files (workingtree.py, mutabletree.py and tree.py) to see which methods are available.

### 3.1 Compare trees

There are two methods for comparing trees: `changes_from` and `iter_changes`. `iter_changes` is more regular and precise, but it is somewhat harder to work with. See the API documentation for more details.

`changes_from` creates a Delta object showing changes:

```
from bzrlib import delta
changes = wt.changes_from(wt.basis_tree())
```

This gives us a Delta object, which has several lists of files for each type of change, eg `changes.added` is a list of added files, `changes.removed` is list of removed files, `changes.modified` is a list of modified files. The contents of the lists aren't just filenames, but include other information as well. To grab just the filename we want the first value, eg:

```
print("list of newly added files")
for filename in changes.added:
    print("%s has been added" % filename[0])
```

The exception to this is `changes.renamed`, where the list returned for each renamed files contains both the old and new names – one or both may interest you, depending on what you're doing.

For example:

```
print("list of renamed files")
for filename in changes.renamed:
    print("%s has been renamed to %s" % (filename[0], filename[1]))
```

### 3.2 Adding Files

If you want to add files the same way `bzr add` does, you can use `MutableTree.smart_add`. By default, this is recursive. Paths can either be absolute or relative to the workingtree:

```
wt.smart_add(['dir1/filea.txt', 'fileb.txt',
             '/home/jebw/bzrtesttree/filec.txt'])
```

For more precise control over which files to add, use `MutableTree.add`:

```
wt.add(['dir1/filea.txt', 'fileb.txt', '/home/jebw/bzrtesttree/filec.txt'])
```

### 3.3 Removing Files

You can remove multiple files at once. The file paths need to be relative to the workingtree:

```
wt.remove(['filea.txt', 'fileb.txt', 'dir1'])
```

By default, the files are not deleted, just removed from the inventory. To delete them from the filesystem as well:

```
wt.remove(['filea.txt', 'fileb.txt', 'dir1'], keep_files=False)
```

### 3.4 Renaming a File

You can rename one file to a different name using `WorkingTree.rename_one`. You just provide the old and new names, eg:

```
wt.rename_one('oldfile.txt', 'newfile.txt')
```

### 3.5 Moving Files

You can move multiple files from one directory into another using `WorkingTree.move`:

```
wt.move(['olddir/file.txt'], 'newdir')
```

More complicated renames/moves can be done with `transform.TreeTransform`, which is outside the scope of this document.

### 3.6 Committing Changes

To commit all the changes to our working tree we can just call the `WorkingTree`'s `commit` method, giving it a commit message, eg:

```
wt.commit('this is my commit message')
```

To commit only certain files, we need to provide a list of filenames which we want committing, eg:

```
wt.commit(message='this is my commit message', specific_files=['fileA.txt',  
    'dir2/fileB.txt', 'fileD.txt'])
```

## 4 Generating a Log for a File

Generating a log is, in itself, simple. Grab a branch (see below) and pass it to `show_log` together with a log formatter, eg:

```
from bzrlib import log  
from bzrlib import branch  
  
b = branch.Branch.open('/path/to/bazaar/branch')  
lf = log.LongLogFormatter(to_file=sys.stdout)  
log.show_log(b, lf)
```

Three log formatters are included with `bzrlib`: `LongLogFormatter`, `ShortLogFormatter` and `LineLogFormatter`. These provide long, short and single-line log output formats. It's also possible to write your own in very little code.

## 5 Annotating a File

To annotate a file, we want to walk every line of a file, retrieving the revision which last modified/created that line and then retrieving the information for that revision.

First we get an annotation iterator for the file we are interested in:

```
tree, relpath = workingtree.WorkingTree.open_containing('/path/to/file.txt')
fileid = tree.path2id(relpath)
annotation = list(tree.annotate_iter(fileid))
```

To avoid repeatedly retrieving the same revisions we grab all revisions associated with the file at once and build up a map of id to revision information. We also build an map of revision numbers, again indexed by the revision id:

```
revision_ids = set(revision_id for revision_id, text in annotation)
revisions = tree.branch.repository.get_revisions(revision_ids)
revision_map = dict(izip(revision_ids, revisions))
revno_map = tree.branch.get_revision_id_to_revno_map()
```

Finally, we use our annotation iterator to walk the lines of the file, displaying the information from our revision maps as we go:

```
for revision_id, text in annotation :
    rev = revision_map[revision_id]
    revno = revno_map[revision_id]
    revno_string = '.'.join(str(i) for i in revno)
    print "%s, %s: %s" % (revno_string, rev.committer, text)
```

## 6 Working with branches

To work with a branch you need a branch object, created from your branch:

```
from bzrlib import branch

b = branch.Branch.open('/home/jebw/bzrtest')
```

### 6.1 Branching from an existing branch

To branch you create a branch object representing the branch you are branching from, and supply a path/url to the new branch location. The following code clones the `bzr.dev` branch (the latest copy of the Bazaar source code) - be warned it has to download 60meg so takes a while to run with no feedback:

```
from bzrlib import branch

b = branch.Branch.open('http://bazaar.launchpad.net/~bzd-pqm/bzr/bzr.dev')
nb = b.bzrdir.sprout('/tmp/newBzrBranch').open_branch()
```

This provides no feedback, since Bazaar automatically uses the 'silent' UI.

### 6.2 Pushing and pulling branches

To push a branch you need to open the source and destination branches, then just call push with the other branch as a parameter:

```
from bzrlib import branch

b1 = branch.Branch.open('file:///home/user/mybranch')
b2 = branch.Branch.open('http://bazaar.launchpad.net/~bzd-pqm/bzr/bzr.dev')
b1.push(b2)
```

Pulling is much the same:

```
b1.pull(b2)
```

If you have a working tree, as well as a branch, you should use `WorkingTree.pull`, not `Branch.pull`.

This won't handle conflicts automatically though, so any conflicts will be left in the working tree for the user to resolve.

## 7 Checkout from an existing branch

This performs a Lightweight checkout from an existing Branch:

```
from bzrlib import bzrdir

accelerator_tree, source = bzrdir.BzrDir.open_tree_or_branch('http:URL')
source.create_checkout('/tmp/newBzrCheckout', None, True, accelerator_tree)
```

To make a heavyweight checkout, change the last line to:

```
source.create_checkout('/tmp/newBzrCheckout', None, False, accelerator_tree)
```

## 8 History Operations

### 8.1 Finding the last revision number or id

To get the last revision number and id of a branch use:

```
revision_number, revision_id = branch.last_revision_info()
```

If all you care about is the `revision_id` there is also the method:

```
revision_id = branch.last_revision()
```

### 8.2 Getting the list of revision ids that make up a branch

**IMPORTANT:** This should be avoided wherever possible, as it scales with the length of history:

```
revisions = branch.revision_history()
```

now `revisions[0]` is the revision id of the first commit, and `revisions[-1]` is the revision id of the most recent. Note that if all you want is the last revision then you should use `branch.last_revision()` as described above, as it is vastly more efficient.

### 8.3 Getting a Revision object from a revision id

The Revision object has attributes like "message" to get the information about the revision:

```
repo = branch.repository
revision = repo.get_revision(rev_id)
```

## 8.4 Accessing the files from a revision

To get the file contents and tree shape for a specific revision you need a `RevisionTree`. These are supplied by the repository for a specific revision id:

```
revtree = repo.revision_tree(rev_id)
```

`RevisionTrees`, like all trees, can be compared as described in “Comparing Trees” above.

The most common way to list files in a tree is `Tree.iter_entries()`. The simplest way to get file content is `Tree.get_file()`. The best way to retrieve file content for large numbers of files `Tree.iter_files_bytes()`