DevAssistant Documentation

Release 0.8.0

Bohuslav Kabrda, Petr Hracek

December 05, 2013

Contents

DevAssistant - making life easier for developers

CHAPTER 1

Contents

1.1 User Documentation

DevAssistant is developer's best friend (right after coffee).

DevAssistant can help you with creating and setting up basic projects in various languages, installing dependencies, setting up environment etc. There are three main types of functionality provided:

- `da crt` - create new project from scratch
- `da mod` - take local project and do something with it (e.g. import it to Eclipse)
- `da prep` - prepare development environment for an upstream project or a custom task

DevAssistant is based on idea of per-{language/framework/...} "assistants" with hierarchical structure. E.g. you can run:

```
$ da crt python django -n ~/myproject # sets up Django project named "myproject" inside your home di
$ da crt python flask -n ~/flaskproject # sets up Flask project named "flaskproject" inside your home
$ da crt ruby rails -n ~/alsomyproject # sets up RoR project named "alsomyproject" inside your home
```

DevAssistant also allows you to work with a previously created project, for example import it to Eclipse:

```
$ da mod eclipse # run in project dir or use -p to specify path
```

With DevAssistant, you can also prepare environment for developing upstream projects - either using project-specific assistants or using "custom" assistant for arbitrary projects (even those not created by DevAssistant):

```
$ da prep custom custom -u scm_url
```

Warning: The `custom` assistant executes custom pieces of code from `.devassistant` file of the project. Therefore you have to be extra careful and use this **only with projects whose authors you trust**.

Last but not least, DevAssistant allows you to perform arbitrary tasks not related to a specific project:

1.1.1 So What is an Assistant?

In short, assistant is a recipe for creating/modifying a project or setting up environment in a certain way. DevAssistant is in fact just a core that "runs" assistants according to certain rules.

Each assistant specifies a way how to achieve a single task, e.g. create a new project in framework X of language Y.

If you want to know more about how this all works, consult *Yaml Assistant Reference*.

Assistant Roles

There are four assistant roles:

creator (`crt` in short) creates new projects

modifier (`mod` in short) modifies existing projects

preparer (`prep` in short) prepares environment for development of upstream projects

task (`task` in short) performs arbitrary tasks not related to a specific project

The main purpose of having roles is separating different types of tasks. It would be confusing to have e.g. `python django` assistant (that creates new project) side-by-side with `eclipse` assistant (that registers existing project into Eclipse).

You can learn about how to invoke the respective roles below in *Creating New Projects*, *Modifying Existing Projects* and *Preparing Environment*.

1.1.2 Using Commandline Interface

Creating New Projects

DevAssistant can help you create (that's the `crt` in the below command) your projects with one line in terminal. For example:

```
$ da crt python django -n foo -e -g
```

`da` is short form of `devassistant`. You can use any of them, but `da` is preferred.

This line will do the following:

- Install Django (RPM packaged) and all needed dependencies.
- Create a Django project named `foo` in current working directory.
- Make any necessary adjustments so that you can run the project and start developing right away.
- The `-e` switch will make DevAssistant register the newly created projects into Eclipse (tries `~/workspace` by default, if you have any other, you need to specify it as an argument to `-e`). This will also cause installation of Eclipse and PyDev, unless already installed.
- The `-g` switch will make DevAssistant register the project on Github and push sources there. DevAssistant will ask you for your Github password the first time you're doing this and then it will create Github API token and new SSH keys, so on any further invocation, this will be fully automatic. Note, that if your system username differs from your Github username, you must specify Github username as an argument to `-g`.

Modifying Existing Projects

DevAssistant allows you to work with previously created projects. You can do this by using `da mod`, as opposed to `da crt` for creating:

```
$ da mod eclipse
```

This will import previously created project into Eclipse (and possibly install Eclipse and other dependencies implied by the project language). Optionally, you can pass `-p path/to/project` if your current working directory is not the project directory.

Preparing Environment

DevAssistant can set up environment and install dependencies for development of already existing project located in a remote SCM (e.g. Github). For custom projects you can use the `custom` assistant. Note that for projects that don't have `.devassistant` file, this will just checkout the sources:

```
$ da prep custom -u scm_url
```

Warning: The `custom` assistant executes custom pieces of code from `.devassistant` file, so use this only for projects whose upstreams you trust.

The plan is to also include assistants for well known and largely developed projects (that, of course, don't contain `.devassistant` file). So in future you should be able to do something like:

```
$ da prep openstack
```

and it should do everything needed to get you started developing OpenStack in a way that others do. But this is still somewhere in the future...

Tasks

The last piece of functionality is performing arbitrary tasks that are not related to a specific projects. E.g.:

```
$ da task <TODO:NOTHING YET>
```

Custom Actions

There are also some custom actions besides `crt`, `mod` and `prep`. For the time being, these are not of high importance, but in future, these will bring more functionality, such as making coffee for you.

`help` Displays help, what else?

`version` Displays current DevAssistant version.

1.1.3 Using GUI

DevAssistant GUI provides the full functionality of *Commandline Interface* through a Gtk based application.

As opposed to CLI, which consists of three binaries, GUI provides all assistant types (creating, modifying, preparing) in one, each type having its own page.

The GUI workflow is dead simple:

- Choose the assistant that you want to use, click it and possibly choose a proper subassistant (e.g. `django` for `python`).
- GUI displays a window where you can modify some settings and choose from various assistant-specific options.
- Click "Run" button and then just watch getting the stuff done. If your input is needed (such as confirming dependencies to install), DevAssistant will ask you, so don't go get your coffee just yet.
- After all is done, get your coffee and enjoy.

1.1.4 Currently Supported Assistants

Please note that list of currently supported assistants may vary greatly in different distributions, depending on available packages etc.

Currently supported assistants with their specialties (if any):

Creating

- C - a simple C project, allows you to create SRPM and build RPM by specifying `-b`
- C++
- Java - JSF - Java Server Faces project - Maven - A simple Apache Maven project
- Perl - Class - Simple class in Perl - Dancer - Dancer framework project
- PHP - LAMP - Apache/MySQL/PHP project
- Python - all Python assistants allow you to use `--venv` switch, which will make DevAssistant create a project inside a Python virtualenv and install dependencies there, rather then installing them system-wide from RPM - Django - Initial Django project, set up to be runnable right away - Flask - A minimal Flask project with a simple view and script for managing the application - Library - A custom Python library - PyGTK - Sample PyGTK project
- Ruby - Rails - Initial Ruby on Rails project

Modifying

- Eclipse - add an existing project into Eclipse (doesn't work for some languages/frameworks)
- Vim - install some interesting Vim extensions and make some changes in `.vimrc` (these changes will not affect your default configuration, instead you have to use command `let devassistant=1` after invoking Vim)

Preparing

- Custom - checkout a custom previously created project from SCM (git only so far) and install needed dependencies

Tasks

<TODO: NOTHING YET>

1.2 Developer Documentation

1.2.1 DevAssistant Core

Note: So far, this only covers some bits and pieces of the whole core.

DevAssistant Load Paths

DevAssistant has couple of load path entries, that are searched for assistants, snippets, icons and files used by assistants. In standard installations, there are three paths:

1. "system" path, which is defined by OS distribution (usually `/usr/share/devassistant/`) or by Python installation (sth. like `/usr/share/pythonX.Y/devassistant/data/`)
2. "local" path, `/usr/local/share/devassistant/`
3. "user" path, `~/.devassistant/`

Another path(s) can be added by specifying `DEVASSISTANT_PATHS` environment variable (if more paths are used, they must be separated by colon). These paths are prepended to the list of standard load paths.

Each load path entry has this structure:

```
assistants/
  crt/
  mod/
  prep/
  task/
files/
  crt/
  mod/
  prep/
  task/
  snippets/
icons/
  crt/
  mod/
  prep/
  task/
snippets/
```

Icons under `icons` directory and files in `files` directory "copy" must the structure of `assistants` directory. E.g. for assistant `assistants/crt/foo/bar.yaml`, the icon must be `icons/crt/foo/bar.svg` and files must be placed under `files/crt/foo/bar/`

Assistants Loading Mechanism

DevAssistant loads assistants from all load paths mentioned above (more specifically from `<load_path>/assistants/` only), traversing them in order "system", "local", "user".

When DevAssistant starts up, it loads all assistants from all these paths. It assumes, that Creator assistants are located under `crt` subdirectories the same applies to Modifier (`mod`), Preparer (`prep`) and Task (`task`) assistants.

For example, loading process for Creator assistants looks like this:

1. Load all assistants located in `crt` subdirectories of each `<load path>/assistants/` (do not descend into subdirectories). If there are multiple assistants with the same name in different load paths, the first traversed wins.
2. For each assistant named `foo.yaml`:
   (a) If `crt/foo` directory doesn't exist in any load path entry, then this assistant is "leaf" and therefore can be directly used by users.
   (b) Else this assistant is not leaf and DevAssistant loads its subassistants from the directory, recursively going from point 1).

Command Runners

Command runners... well, they run commands. They are the functionality that makes DevAssistant powerful, since they effectively allow you to create callbacks to Python, where you can cope with the hard parts unsuitable for Yaml assistants.

When DevAssistant executes a `run` section, it reads commands one by one and dispatches them to their respective command runners. Every command runner can do whatever it wants - for example, we have a command runner that creates Github repos.

After a command runner is run, DevAssistant sets `LAST_LRES` and `LAST_RES` global variables for usage (these are rewritten with every command run). These variables represent the logical result of the command (`True`/`False`) and result (a "return value", something computed), much like with *Expressions*.

For reference of current commands, see *Command Reference*.

If you're missing some cool functionality, you can implement your own command runner and send us a pull request. (We're thinking of creating some sort of import hook that would allow assistants to import command runners from Python files outside of DevAssistant, but it's not on the priority list right now.) Each command must be a class with two classmethods:

```
@register_command_runner
class MyCommandRunner(CommandRunner):
    @classmethod
    def matches(cls, c):
        return c.comm_type == 'mycomm'

    @classmethod
    def run(cls, c):
        formatted = c.format_str()
        logger.info('MyCommandRunner was invoked: {ct}: {ci}'.format(ct=c.comm_type,
                                                                      ci=formatted))
        return [True, len(formatted)]
```

This command runner will run all commands with command type `mycomm`. For example if your assistant contains:

```
run:
- $foo: $(echo "using DevAssistant")
- mycomm: You are $foo!
```

than DevAssistant will print out something like:

```
INFO: MyCommandRunner was invoked: mycomm: You are using DevAssistant!
```

After this command is run, `LAST_LRES` will be set to `True` and `LAST_RES` to length of the printed string.

Generally, the `matches` method should just decide (True/False) whether given command is runnable or not and the `run` method should actually run it. The `run` method should use devassistant.logger.logger object to log any messages and it can also raise any exception that's subclass of `devassistant.exceptions.ExecutionException`.

The `c` argument of both methods is a `devassistant.command.Command` object. You can access the **command type** via `c.comm_type` and raw **command input** via `c.comm`. If you want to get input as a formatted string, where variables are substituted for their values, use `c.format_str()`. You can also access (and change - use this wisely!) the global mapping of variables via `c.kwargs`.

1.2.2 Tutorial: Creating Your Own Assistant

So you want to create your own assistant? There is nothing easier... They say that in all tutorials, right?

This tutorial will guide you through the process of creating simple assistants of *different roles* - Creator, Modifier, Preparer.

This tutorial doesn't cover everything. Consult *Yaml Assistant Reference* when you're missing something you really need to achieve. If you think that DevAssistant misses some functionality that would be useful, open a bug at https://www.github.com/bkabrda/devassistant/issues or send us a pull request.

Common Rules and Gotchas

Some things are common for all assistant types:

- Each assistant is one Yaml file, that must contain exactly one mapping of assistant name to all the assistant contents. E.g:

  ```
  assistant:
    fullname: My Assistant
    description: This will be part of help for this assistant
    ...
  ```

- You have to place them in a proper place, see *DevAssistant Load Paths* and *Assistants Loading Mechanism*.
- When creating templates (pre-created files used by assistants), they should be placed in the same load dir, e.g. if your assistant is placed at `~/.devassistant/assistants`, it will look for templates under `~/.devassistant/templates`.
- As mentioned in *DevAssistant Load Paths*, there are three main load paths in standard DevAssistant installation, "system", "local" and "user". The "system" dir is used for assistants delivered by your distribution/packaging system and you shouldn't touch or add files in this path. The "local" path can be used by system admins to add system-wide assistants while not touching "system" path. Lastly, "user" path can be used by users to create and use their own assistants. It is up to you where you place your assistant, but "user" path is usually best for playing around and development of new assistants. It is also the path that we will use throughout these tutorials.

Creating a Simple Creator

The title says it all. In this section, we will create a "Creator" assistant, that means an assistant that will take care of kickstarting a new project. We will write an assistant that creates a project containing a simple Python script that uses `argh` Python module. Let's suppose that we're writing this assistant for an RPM based system like Fedora, CentOS or RHEL.

This assistant is a "creator", so we have to put it somewhere into `~/.devassistant/assistants/crt/`. Since the standard DevAssistant distribution has a `python` assistant, it seems logical to make this new assistant a subassistant of `python`. That means that the assistant file will be `~/.devassistant/assistants/creator/python/argh.yaml`. It doesn't matter that the `python` assistant actually lives in a different load path, DevAssistant will hook the `argh` subassistant properly anyway.

Setting it Up

So, let's start writing our assistant by providing some initial metadata:

```
argh:
  fullname: Argh Script Template
  description: Create a template of simple script that uses argh library
```

If you now save the file and run `da crt python argh -h`, you'll see that your assistant was already recognized by DevAssistant, although it doesn't provide any functionality yet.

Dependencies

Now, we'll want to add a dependency on `python-argh` (which is how the package is called e.g. on Fedora). You can do this just by adding:

```
dependencies:
- rpm: [python-argh]
```

Now, if you save the file and actually try to run your assistant with `da crt python argh`, it will install `python-argh`! (Well, assuming it's not already installed, in which case it will do nothing.) This is really super-cool, but the assistant still doesn't do any project setup, so let's get on with it.

Files

Since we want the script to always look the same, we will create a file that our assistant will copy into proper place. This file should be put into into `crt/python/argh` subdirectory the template directory (`~/.devassistant/files/crt/python/argh`). The file will be called `arghscript.py` and will have this content:

```
#!/usr/bin/python2

from argh import *

def main():
    return 'Hello world'

dispatch_command(main)
```

We will need to refer to this file from our assistant, so let's open `argh.yaml` again and add a `files` section:

```
files:
  arghs: &arghs
    source: arghscript.py
```

DevAssistant will automatically search for this file in the correct directory, that is `~/.devassistant/files/crt/python/argh`. If there are e.g. some files common to multiple `python` subassistants, it is reasonable to place them into `~/.devassistant/files/crt/python` and refer to them with relative path like `../file.foo`

Run

Finally, we will be adding a `run` section, which is the section that does all the hard work. A `run` section is a list of **commands**. Every command is in fact a Yaml mapping with exactly one key and value. The key determines **command type**, while value is the **command input**. For example, `cl` is a **command type** that says that given **input** should be run on commandline, `log_i` is a **command type** that lets us print the **input** (message in this case) for user, etc.

Let's start writing our `run` section:

```
run:
- log_i: Hello, I'm Argh assistant and I will create an argh project for you.
```

But wait! We don't know what the project should be called and where it should be placed... Before we finish the `run` section, we'll need to add some arguments to our assistant.

Oh Wait, Arguments!

Creating any type of project typically requires some user input, at least name of the project to be created. To ask user for this sort of information, we can use DevAssistant arguments like this:

```
args:
  name:
    flags: [-n, --name]
    required: True
    help: 'Name of project to create'
```

This means that this assistant will have one argument called `name`. On commandline, it will expect `-n foo` or `--name foo` and since the argument is required, it will refuse to run without it.

You can now try running `da crt python argh -h` and you'll see that the argument is printed out in commandline help.

Since there are some common arguments, the standard installation of DevAssistant ships with so called "snippets", that contain (among other things) definitions of frequentyl used arguments. You can use name argument for Creator assistants like this:

```
args:
  name:
    use: common_args
```

Note: up to version 0.8.0, "snippet" can also be used in place of "use"; "snippet" is obsolete and will be removed in 0.9.0.

Run Again

Now that we can obtain the desired name, let's continue. Now that we have the project name (let's assume that it's an arbitrary path to a directory where the argh script should be placed), we can continue. First, we will make sure that the directory doesn't already exist. If so, we need to exit, because we don't want to overwrite or break something:

```
run:
- log_i: Hello, I'm Argh assistant and I will create an argh project for you.
- if $(test -e "$name"):
  - log_e: '"$name" already exists, can't proceed.'
```

There are few things to note here:

- There is a simple `if` condition with a shell command. If the shell command returns a non-zero value, the condition will evaluate to false, else it will evaluate to true. So in this case, if something exists at path `"$name"`, the condition will evaluate to true.
- In any command, we can use value of the `name` argument by prefixing argument name with `$` (so `$name` or `${name}`).
- The `log_e` command type is used to print a message and then abort the assistant execution immediately.

Let's continue by creating the directory. Add this line to `run` section:

```
- cl: mkdir -p "$name"
```

You may be wondering what will happen, if DevAssistant doesn't have write permissions or more generally if the `mkdir` command just fails. In this case, DevAssistant will exit, printing the output of failed command for user.

Next, we want to copy our script into the directory. We want to name it the same as name of the directory itself. But what if directory is a path, not simple name? We have to find out the project name and remember it somehow:

```
- $proj_name: $(basename "$name")
```

What just happened? We assigned output of command `basename "$name"` to a new variable `proj_name` that we can use from now on. So let's copy the script and make it executable:

```
- cl: cp *arghs ${name}/${proj_name}.py
- cl: chmod +x ${name}/${proj_name}.py
```

One thing to note here is, that by using `*arghs`, we reference a file from the `files` section.

Now, we'll use a super-special command:

```
- dda_c: "$name"
```

What is `dda_c`? The first part, `dda` stands for "dot devassistant file", the second part, `_c`, says, that we want to create this file (there are more things that can be done with `.devassistant` file, see TODO). The "command" part of this call just says where the file should be stored, which is `$name` directory in our case.

The `.devassistant` file serves for storing meta information about the project. Amongst other things, it stores information about which assistant was invoked. This information can later serve to prepare the environment (e.g. install `python-argh`) on another machine or so. Assuming that we commit the project to a git repository, one just needs to run `da prep custom -u <repo_url>`, and DevAssistant will checkout the project from git and use information stored in `.devassistant` to reinstall dependencies. (There is more to this, you can for example add a custom `run` section to `.devassistant` file or add custom dependencies, but this is not covered by this tutorial (not even by reference, so I need to place TODO here to document it).)

Note: There can be more dependencies sections and run sections in one assistant. To find out more about the rules of when they're used and how run sections can call each other, consult dependencies reference and run reference.

Something About Snippets

Wait, did we say git? Wouldn't it be nice if we could setup a git repository inside the project directory and do an initial commit? These things are always the same, which is exactly the type of task that DevAssistant should do for you.

Previously, we've seen usage of argument from snippet. But what if you could use a part of `run` section from there? Well, you can. And you're lucky, since there is a snippet called `git_init_add_commit`, which does exactly what we need. We'll use it like this:

```
- cl: cd "$name"
- use: git_init_add_commit
```

This calls section `run` from snippet `git_init_add_commit` in this place. Note, that all variables are "global" and the snippet will have access to them and will be able to change their values. However, variables defined in called snippet section will not propagate into current section.

Note: up to version 0.8.0, "call" can also be used in place of "use"; "call" is obsolete and will be removed in 0.9.0.

Finished!

It seems that everything is set. It's always nice to print a message that everything went well, so we'll do that and we're done:

```
- log_i: Project "$proj_name" has been created in "$name".
```

The Whole Assistant

... looks like this:

```
argh:
  fullname: Argh Script Template
  description: Create a template of simple script that uses argh library

  dependencies:
  - rpm: [python-argh]

  files:
    arghs: &arghs
      source: arghscript.py

  args:
    name:
      use: common_args

  run:
  - log_i: Hello, I'm Argh assistant and I will create an argh project for you.
  - if $(test -e "$name"):
    - log_e: '"$name" already exists, cannot proceed.'
  - cl: mkdir -p "$name"
  - $proj_name: $(basename "$name")
  - cl: cp *arghs ${name}/${proj_name}.py
  - cl: chmod +x *arghs ${name}/${proj_name}.py
  - dda_c: "$name"
  - cl: cd "$name"
  - use: git_init_add_commit
  - log_i: Project "$proj_name" has been created in "$name".
```

And can be run like this: `da crt python argh -n foo/bar`.

Creating a Modifier

This section assumes that you've read the previous tutorial and are therefore familiar with DevAssistant basics. Modifiers are meant to modify existing projects, that means projects with `.devassistant` file (there is also an option to write assistant that modifies an arbitrary project without `.devassistant`, read on).

Modifier Specialties

The special behaviour of modifiers only applies if you use dda_r in pre_run section. This command reads .devassistant file from given directory and puts the read variables in global variable context, so they're available from all the following dependencies and run section.

If modifier reads `.devassistant` file in `pre_run` section, DevAssistant tries to search for more `dependencies` sections to use. If the project was previously created by `crt python django`, the engine will install dependencies from sections `dependencies_python_django`, `dependencies_python` and `dependencies`.

Also, the engine will try to run `run_python_django` section first, then it will try `run_python` and then `run` - note, that this will only run the first found section and then exit, unlike with dependencies, where all found sections are used.

– IN PROGRESS –

1.2.3 Yaml Assistant Reference

When developing assistants, please make sure that you read proper version of documentation. The Yaml DSL of devassistant is still evolving rapidly, so consider yourself warned.

This is a reference manual to writing yaml assistants. Yaml assistants use a special DSL defined on this page. For real examples, have a look at assistants in our Github repo.

Why the hell another DSL? When we started creating DevAssistant and we were asking people who work in various languages whether they'd consider contributing assistants for those languages, we hit the "I'm not touching Python" barrier. Since we wanted to keep the assistants consistent (centralized logging, sharing common functionality, same backtraces, etc...), we created a new DSL. So now we have something that everyone complains about, including Pythonists, which seems to be consistent too ;)

Assistant Roles

For list and description of assistant roles see *Assistant Roles*.

The role is implied by assistant location in one of the load path directories, as mentioned in *Assistants Loading Mechanism*.

All the rules mentioned in this document apply to all types of assistants, with exception of sections *Modifier Assistants*, *Preparer Assistants* and *Task Assistants* that talk about specifics of Modifier, resp. Preparer, resp. Task assistants.

Assistant Name

Assistant name is a short name used on command line, e.g. `python`. It should also be the only top-level yaml mapping in the file (that means just one assistant per file). Each assistant should be placed in a file that's named the same as the assistant itself (e.g. `python` assistant in `python.yaml` file).

Assistant Content

The top level mapping has to be mapping from assistant name to assistant attributes, for example:

```
python:
  fullname: Python
  # etc.
```

List of allowed attributes follows (all of them are optional, and have some sort of reasonable default, it's up to your consideration which of them to use):

`fullname` a verbose name that will be displayed to user (`Python Assistant`)

`description` a (verbose) description to show to user (`Bla bla create project bla bla`)

`dependencies` (and `dependencies_*`) specification of dependencies, see below Dependencies

`args` specification of arguments, see below Args

`files` specification of used files, see below Files

`run` (and `run_*`) specification of actual operations, see below Run

`files_dir` directory where to take files (templates, helper scripts, ...) from. Defaults to base directory from where this assistant is taken + `files`. E.g. if this assistant is `~/.devassistant/assistants/crt/path/and/more.yaml`, files will be taken from `~/.devassistant/files/crt/path/and/more` by default.

icon_path absolute or relative path to icon of this assistant (will be used by GUI). If not present, a default path will be used - this is derived from absolute assistant path by replacing `assistants` by `icons` and `.yaml` by `.svg` - e.g. for `~/.devassistant/assistants/crt/foo/bar.yaml`, the default icon path is `~/.devassistant/icons/crt/foo/bar.svg`

Assistants Invocation

When you invoke DevAssistant with it will run following assistants sections in following order:

- `pre_run`
- `dependencies`
- `run` (possibly different section for Modifier Assistants)
- `post_run`

If any of the first three sections fails in any step, DevAssistant will immediately skip to `post_run` and the whole invocation will be considered as failed (will return non-zero code on command line and show "Failed" in GUI).

Dependencies

Yaml assistants can express their dependencies in multiple sections.

- Packages from section `dependencies` are **always** installed.
- If there is a section named `dependencies_foo`, then dependencies from this section are installed **iff** `foo` argument is used (either via commandline or via gui). For example:

  ```
  $ da python --foo
  ```

- These rules differ for Modifier Assistants

Each section contains a list of mappings `dependency type: [list, of, deps]`. If you provide more mappings like this:

```
dependencies:
- rpm: [foo]
- rpm: ["@bar"]
```

they will be traversed and installed one by one. Supported dependency types:

rpm the dependency list can contain RPM packages or YUM groups (groups must begin with `@` and be quoted, e.g. `"@Group name"`)

use / call (these two do completely same, call is obsolete and will be removed in 0.9.0) installs dependencies from snippet/another dependency section of this assistant/dependency section of superassistant. For example:

```
dependencies:
- use: foo # will install dependencies from snippet "foo", section "dependencies"
- use: foo.dependencies_bar # will install dependencies from snippet "foo", section "bar"
- use: self.dependencies_baz # will install dependencies from section "dependencies_baz" of this
- use: super.dependencies # will install dependencies from "dependencies" section of first super
```

if, else conditional dependency installation. For more info on conditions, Run below. A very simple example:

```
dependencies:
- if $foo:
  - rpm: [bar]
```

```
  - else:
    - rpm: [spam]
```

Full example:

```
dependencies: - rpm: [foo, "@bar"]

dependencies_spam:
- rpm: [beans, eggs]
- if $with_spam:
  - use: spam.spamspam
- rpm: ["ham${more_ham}"]
```

Sometimes your dependencies may get terribly complex - they depend on many parameters, you need to use them dynamically during ``run``, etc. In these cases, it is better to use ``dependencies`` command in ``run`` section.

Args

Arguments are used for specifying commandline arguments or gui inputs. Every assistant can have zero to multiple arguments.

The `args` section of each yaml assistant is a mapping of arguments to their attributes:

```
args:
  name:
    flags:
    - -n
    - --name
  help: Name of the project to create.
```

Available argument attributes:

`flags` specifies commandline flags to use for this argument. The longer flag (without the `--`, e.g. `name` from `--name`) will hold the specified commandline/gui value during `run` section, e.g. will be accessible as `$name`.

`help` a help string

`required` one of `{true,false}` - is this argument required?

`nargs` how many parameters this argument accepts, one of `{?,*,+}` (e.g. {0 or 1, 0 or more, 1 or more})

`default` a default value (this will cause the default value to be set even if the parameter wasn't used by user)

`action` one of `{store_true, [default_iff_used, value]}` - the `store_true` value will create a switch from the argument, so it won't accept any parameters; the `[default_iff_used, value]` will cause the argument to be set to default value `value` **iff** it was used without parameters (if it wasn't used, it won't be defined at all)

`use` / `snippet` (these two do completely same, `snippet` is obsolete and will be removed in 0.9.0) name of the snippet to load this argument from; any other specified attributes will override those from the snippet By convention, some arguments should be common to all or most of the assistants. See *Common Assistant Behaviour*

Gui Hints

GUI needs to work with arguments dynamically, choose proper widgets and offer sensible default values to user. These are not always automatically retrieveable from arguments that suffice for commandline. For example, GUI cannot meaningfully prefill argument that says it "defaults to current working directory". Also, it cannot tell whether to choose a widget for path (with the "Browse ..." button) or just a plain text field.

Because of that, each argument can have `gui_hints` attribute. This can specify that this argument is of certain type (path/str/bool) and has a certain default. If not specified in `gui_hints`, the default is taken from the argument itself, if not even there, a sensible "empty" default value is used (home directory/empty string/false). For example:

```
args:
  path:
    flags:
    - [-p, --path]
    gui_hints:
      type: path
      default: $(pwd)/foo
```

If you want your assistant to work properly with GUI, it is good to use `gui_hints` (currently, it only makes sense to use it for `path` attributes, as `str` and `bool` get proper widgets and default values automatically).

Files

This section serves as a list of aliases of files stored in one of the `files` dirs of DevAssistant. E.g. if your assistant is `assistants/crt/foo/bar.yaml`, then files are taken relative to `files/crt/foo/bar/` directory. So if you have a file `files/crt/foo/bar/spam`, you can use:

```
files:
  spam: &spam
    source: spam
```

This will allow you to reference the `spam` file in `run` section as `*spam` without having to know where exactly it is located in your installation of DevAssistant.

Run

Run sections are the essence of DevAssistant. They are responsible for preforming all the tasks and actions to set up the environment and the project itself. For Creator and Preparer assistants, section named `run` is always invoked, Modifier Assistants may invoke different sections based on metadata in `.devassistant` file.

Note, that `pre_run` and `post_run` follow the same rules as `run` sections. See Assistants Invocation to find out how these sections are invoked.

Every `run` section is a sequence of various **commands**, mostly invocations of commandline. Each command is a mapping of **command type** to **command input**:

```
run:
- cl: cp foo bar/baz
- log_i: Done copying.
```

During the execution, you may use logging (messages will be printed to terminal or gui) with following levels: `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`. By default, messages of level `INFO` and higher are logged. As you can see below, there is a separate `log_*` **command** type for logging, but some other command types also log various messages. Log messages with levels `ERROR` and `CRITICAL` terminate execution of DevAssistant imediatelly.

Run sections allow you to use variables with certain rules and limitations. See below.

List of supported **commands** can be found at *Command Reference*.

Variables

Initially, variables are populated with values of arguments from commandline/gui and there are no other variables defined for creator assistants. For modifier assistants global variables are prepopulated with some values read from

`.devassistant`. You can either define (and assign to) your own variables or change the values of current ones.

Additionally, after each command, variables `$LAST_RES` and `$LAST_LRES` are populated with result of the last command - see *Command Reference*

The variable scope works as follows:

- When invoking `run` section (from the current assistant or snippet), the variables get passed by value (e.g. they don't get modified for the remainder of this scope).
- As you would probably expect, variables that get modified in `if` and `else` sections are modified until the end of the current scope.

All variables are global in the sense that if you call a snippet or another section, it can see all the arguments that are defined.

Expressions

Expressions are used in assignments, conditions and as loop "iterables". Every expression has a *logical result* (meaning success - `True` or failure - `False`) and *result* (meaning output). *Logical result* is used in conditions and variable assignments, *result* is used in variable assignments and loops. Note: when assigned to a variable, the *logical result* of an expression can be used in conditions as expected; the *result* is either `True` or `False`.

Syntax and semantics:

- `$foo`
  - if `$foo` is defined:
    - *logical result*: `True` **iff** value is not empty and it is not `False`
    - *result*: value of `$foo`
  - otherwise:
    - *logical result*: `False`
    - *result*: empty string
- `$(commandline command)` (yes, that is a command invocation that looks like running command in a subshell)
  - if `commandline command` has return value 0:
    - *logical result*: `True`
  - otherwise:
    - *logical result*: `False`
  - regardless of *logical result*, *result* always contains both stdout and stderr lines in the order they were printed by `commandline command`
- `defined $foo` - works exactly as `$foo`, but has *logical result* `True` even if the value is empty or `False`
- `not $foo` negates the *logical result* of an expression, while leaving *result* intact
- `$foo and $bar`
  - *logical result* is logical conjunction of the two arguments
  - *result* is empty string if at least one of the arguments is empty, or the latter argument
- `$foo or $bar`
  - *logical result* is logical disjunction of the two arguments

    - *result* is the first non-empty argument or an empty string

- `literals - "foo", 'foo'`
    - *logical result* `True` for non-empty strings, `False` otherwise
    - *result* is the string itself, sans quotes
    - *Note: If you use an expression that is formed by just a literal, e.g.* `"foo"` *, then DevAssistant will fail, since Yaml parser will strip these. Therefore you have to use* `'"foo"'` *.*
- `$foo in $bar`
    - *logical result* is `True` if the result of the second argument contains the result of the second argument (e.g. “inus” in “Linus Torvalds”) and `False` otherwise
    - *result* is always the first agument

All these can be chained together, so, for instance, `"1.8.1.4" in $(git --version) and defined $git` is also a valid expression

Quoting

When using variables that contain user input, they should always be quoted in the places where they are used for bash execution. That includes `cl*` commands, conditions that use bash return values and variable assignment that uses bash.

Modifier Assistants

Modifier assistants are assistants that are supposed to work with already created project. They must be placed under `mod` subdirectory of one of the load paths, as mentioned in *Assistants Loading Mechanism*.

There are few special things about modifier assistants:

- They usually utilize `dda_r` to read the whole `.devassistant` file (usually from directory specified by `path` variable or from current directory). **Since version 0.8.0, every modifier assistant has to do this on its own, be it in pre_run or run section**. This also allows you to modify non-devassistant projects - just don’t use `dda_r`.

The special rules below **only apply if you use dda_t in pre_run section**.

- They use dependency sections according to the normal rules + they use *all* the sections that are named according to loaded `$subassistant_path`, e.g. if `$subassistant_path` is `[foo, bar]`, dependency sections `dependencies`, `dependencies_foo` and `dependencies_foo_bar` will be used as well as any sections that would get installed according to specified parameters. The rationale behind this is, that if you have e.g. `eclipse` modifier that should work for both `python django` and `python flask` projects, chance is that they have some common dependencies, e.g. `eclipse-pydev`. So you can just place these common dependencies in `dependencies_python` and you’re done (you can possibly place special per-framework dependencies into e.g. `dependencies_python_django`).
- By default, they don’t use `run` section. Assuming that `$subassistant_path` is `[foo, bar]`, they first try to find `run_foo_bar`, then `run_foo` and then just `run`. The first found is used. If you however use cli/gui parameter `spam` and section `run_spam` is present, then this is run instead.

Preparer Assistants

Preparer assistants are assistants that are supposed to checkout sources of upstream projects and set up environment for them (possibly utilizing their `.devassistant` file, if they have one). Preparers must be placed under `prep` subdirectory of one of the load paths, as mentioned in *Assistants Loading Mechanism*.

Preparer assistants commonly utilize the `dda_dependencies` and `dda_run` commands in `run` section.

Task Assistants

Task assistants are supposed to carry out arbitrary task that are not related to a specific project. <TODO>

1.2.4 Command Reference

This page serves as a reference commands of DevAssistant Yaml DSL. Every command consists of **command_type** and **command_input** and sets `LAST_LRES` and `LAST_RES` variables. These two should represent (similarly to *Expressions* **logical result** and **result**):

- `LAST_LRES` - a logical result of the run - `True`/`False` if successful/unsuccessful
- `LAST_RES` - a "return value" - e.g. a computed value

In the Yaml DSL, commands are called like this:

```
command_type: command_input
```

This reference summarizes commands included in DevAssistant itself in following format:

`command_type` - some optional info

- Input: what should the input look like?
- RES: what is `LAST_RES` set to after this command?
- LRES: what is `LAST_LRES` set to after this command?
- Example: example usage

Missing something? Commands are your entrypoint for extending DevAssistant. If you're missing some functionality in `run` sections, just *write a command runner* and send us a pull request.

Builtin Commands

There are three builtin commands that are inherent part of DevAssistant Yaml DSL:

- variable assignment
- condition
- loop

All of these builtin commands utilize expressions in some way - these must follow rules in *Expressions*.

Variable Assignment

Assign *result* (and possibly also *logical result*) of *Expressions* to a variable(s).

`$<var1>[, $<var2>]` - if one variable is given, *result* of expression (**command input**) is assigned. If two variables are given, the first gets assigned *logical result* and the second *result*.

- Input: an expression
- RES: *result* of the expression
- LRES: *logical result* of the expression

- Example:

  ```
  $foo: "bar"
  $spam:
  - spam
  - spam
  - spam
  $bar: $baz
  $success, $list: $(ls "$foo")
  ```

Condition

Conditional execution.

`if <expression>`, `else` - conditionally execute one or the other section (`if` can stand alone, of course)

- Input: a subsection to run
- RES: RES of last command in the subsection, if this clause is invoked. If not invoked, there is no RES.
- LRES: LRES of last command in the subsection, if this clause is invoked. If not invoked, there is no LRES.
- Example:

  ```
  if defined $foo:
  - log_i: Foo is $foo!
  else:
  - log_i: Foo is not defined!
  ```

Loop

A simple for loop.

`for <var>[, <var>] in <expression>` - loop over result of the expression (strings are split in whitespaces). When iterating over mapping, two control variables may be provided to get both key and its value.

- Input: a subsection to repeat in loop
- RES: RES of last command of last iteration in the subsection. If there are no interations, there is no RES.
- LRES: LRES of last command of last iteration in the subsection. If there are no interations, there is no RES.
- Example:

  ```
  for $i in $(ls):
  - log_i: $i

  $foo:
    1: one
    2: two
  for $k, $v in $foo:
  - log_i: $k, $v
  ```

Ask Commands

User interaction commands, let you ask for password and various other input.

`ask_password`

- Input: list of
  - *variable* that gets assigned the password (empty if user denies)
  - mapping containing `prompt` (short prompt for user)
- RES: the password
- LRES: always `True`
- Example:

```
ask_password:
- $passwd
- prompt: "Please provide your password"
```

`ask_confirm`

- Input: list of
  - *variable* that gets assigned the confirmation (`True`/`False`)
  - mapping containing `prompt` (short prompt for user) and `message` (a longer description of what the user should confirm
- RES: the confirmation
- LRES: always `True`
- Example:

```
ask_confirm:
- $confirmed
- message: "Do you think DevAssistant is great?"
  prompt: "Please select yes."
```

Command Line Commands

Run commands in subprocesses and receive their output.

`cl`, `cl_i` (these do the same, but the second version logs the command output on INFO level, therefore visible to user by default)

- Input: a string, possibly containing variables and references to files
- RES: stdout + stdin interleaved as they were returned by the executed process
- LRES: always `True` (if the command fails, the whole DevAssistant execution fails)
- Example:

```
cl: mkdir ${name}
cl: cp *file ${name}/foo
```

Dependencies Command

Install dependencies from given **command input**.

`dependencies`

- Input: list of mappings, similar to *Dependencies section*, but without conditions and usage of sections from snippets etc.

- RES: always `True` (terminates DevAssistant if dependency installation fails)
- LRES: **command input**, but with expanded variables
- Example:

```
if $foo:
- $rpmdeps: [foo, bar]
else:
- $rpmdeps: []

dependencies:
- rpm: $rpmdeps
```

.devassistant Commands

Commands that operate with `.devassistant` file.

`dda_c` - creates a `.devassistant` file, should only be used in creator assistants

- Input: directory where the file is supposed to be created
- RES: always `True`, terminates DevAssistant if something goes wrong
- LRES: always empty string
- Example:

```
dda_c: ${path}/to/project
```

`dda_r` - reads an existing `.devassistant` file, should be used by modifier and preparer assistants.Sets some global variables accordingly, most importantly `original_kwargs` (arguments used when the project was created) - these are also made available with `dda__` prefix (yes, that's double underscore).

- Input: directory where the file is supposed to be
- RES: always `True`, terminates DevAssistant if something goes wrong
- LRES: always empty string
- Example:

```
dda_r: ${path}/to/project
```

`dda_w` - writes a mapping (dict in Python terms) to `.devassistant`

- Input: list with directory with `.devassistant` file as a first item and the mapping to write as the second item. Variables in the mapping will be substituted, you have to use `$$foo` (two dollars instead of one) to get them as variables in `.devassistant`.
- RES: always `True`, terminates DevAssistant if something goes wrong
- LRES: always empty string
- Example:

```
dda_w:
- ${path}/to/project
- run:
  - $$foo: $name # name will get substituted from current variable
  - log_i: $$foo
```

`dda_dependencies` - installs dependencies from `.devassistant` file, should be used by preparer assistants. Utilizes both dependencies of creator assistants that created this project plus dependencies from `dependencies` section, if present (this section is evaluated in the context of current assistant, not the creator).

- Input: directory where the file is supposed to be
- RES: always `True`, terminates DevAssistant if something goes wrong
- LRES: always empty string
- Example:

  ```
  dda_dependencies: ${path}/to/project
  ```

`dda_run` - run `run` section from from `.devassistant` file, should be used by preparer assistants. This section is evaluated in the context of current assistant, not the creator.

- Input: directory where the file is supposed to be
- RES: always `True`, terminates DevAssistant if something goes wrong
- LRES: always empty string
- Example:

  ```
  dda_run: ${path}/to/project
  ```

Jinja2 Render Command

Render a Jinja2 template.

`jinja_render`

- Input: a mapping containing
  - `template` - a reference to file in `files` section
  - `destination` - where to place rendered template
  - `data` - a mapping of values used to render the template itself
  - `overwrite` (optional) - overwrite the file if it exists?
  - `output` (optional) - specify a filename of the rendered template
- RES: always `True`, terminates DevAssistant if something goes wrong
- LRES: always `success` string
- Example:

  ```
  jinja_render:
    template: *somefile
    destination: ${dest}/foo
    overwrite: yes
    output: filename.foo
    data:
      foo: bar
      spam: spam
  ```

The filename of the rendered template is created in this way:

- if `output` is provided, use that as the filename
- else if name of the template endswith `.tpl`, strip `.tpl` and use it

- else use the template name

Logging Commands

Log commands on various levels. Logging on ERROR or CRITICAL logs the message and then terminates the execution.

`log_[d,i,w,e,c]` (the letters stand for DEBUG, INFO, WARNING, ERROR, CRITICAL)

- Input: a string, possibly containing variables and references to files
- RES: the logged message (with expanded variables and files)
- LRES: always `True`
- Example:

```
log_i: Hello $name!
log_e: Yay, something has gone wrong, exiting.
```

SCL Command

Run subsection in SCL environment.

`scl [args to scl command]` (note: you **must** use the scriptlet name - usually `enable` - because it might vary)

- Input: a subsection
- RES: RES of the last command in the given section
- LRES: LRES of the last command in the given section
- Example:

```
- scl enable python33 postgresql92:
  - cl_i: python --version
  - cl_i: pgsql --version
```

Using Another Section

Runs a section specified by **command input** at this place.

`use`, `call` (these two do completely same, `call` is obsolete and will be removed in 0.9.0) This can be used to run:

- another section of this assistant (e.g. `use: self.run_foo`)
- section of superassistant (e.g. `use: super.run`) - searches all superassistants (parent of this, parent of the parent, etc.) and runs the first found section of given name
- section from snippet (e.g. `use: snippet_name.run_foo`)
- Input: a string with section name
- RES: RES of the last command in the given section
- LRES: LRES of the last command in the given section
- Example:

```
    use: self.run_foo
    use: super.run
    use: a_snippet: run_spam
```

1.2.5 Common Assistant Behaviour

Common Parameters of Assistants and Their Meanings

-e Create Eclipse project, optional. Should create `.project` (or any other appropriate file) and register project to Eclipse workspace (`~/workspace` by default, or the given path if any).

-g Register project on GitHub (uses current user name by default, or given name if any).

-n Name of the project to create, mandatory. Should also be able to accept full or relative path.

To include these parameters in your assistant with common help strings etc., include them from `common_args.yaml` (`-n`, `-g`) or `eclipse.yaml` (`-e`) snippet:

```
args:
  name:
    snippet: common_args
```

Other Conventions

When creating snippets/Python commands, they should operate under the assumption that current working directory is the project directory (not one dir up or anywhere else). It is the duty of assistant to switch to that directory. The benefit of this approach is that you just *cd* once in assistant and then call all the snippets/commands, otherwise you'd have to put 2x'cd' in every snippet/command.

1.2.6 Project Metainfo: the .devassistant File

Each project created by DevAssistant gets a `.devassistant` file. This file contains information about the project, such as used Creator assistant or given paramaters. It can look like this:

```
devassistant_version: 0.7.0
original_kwargs:
  name: foo
  github: bkabrda
subassistant_path:
- python
- django
```

When .devassistant is used

Generally, there are two use cases for `.devassistant`:

- Modifier assistants read the `.devassistant` file to get project type (which is specified by `subassistant_path` entry) and decide what to do with this type of project (by choosing a proper `run` section to execute, see *Modifier Assistants*).
- When you use the `custom` preparer with URL to this project (`da prep custom -u <url>`), DevAssistant will checkout the project, read the data from `.devassistant` and install dependencies according to specified `subassistant_path`, assuming the local copy of DevAssistant knows given assistants (e.g. if

your installation of DevAssistant doesn't have `python` or `django` assistant, DevAssistant will just print a warning, but won't install dependencies for those).

Another nice thing about `custom` assistant is, that it will install any dependendencies that it finds in `.devassistant`. These dependencies look like normal *dependencies section* in assistant, e.g.:

```
dependencies:
- rpm: [python-spam]
```

It will also run a `run` section from `.devassistant`, if it is there. Again, this is a normal *run section*:

```
run:
- log_i: Hey, I'm running from .devassistant after checkout!
```

Generally, when using `custom` assistant, you have to be **extra careful**, since someone could put `rm -rf ~` or similar evil command in the `run` section. So use it **only with projects whose upstream you trust**.

1.2.7 Overall Design

DevAssistant consists of several parts:

Core Core of DevAssistant is written in Python. It is responsible for interpreting Yaml Assistants and it provides an API that can be used by any consumer for the interpretation.

CL Interface CL interface allows users to interact with DevAssistant on commandline; it consumes the Core API.

GUI (work in progress) GUI allows users to interact with Developer Assistant from GTK based GUI; it consumes the Core API.

Assistants Assistants are Yaml files with special syntax and semantics (defined in *Yaml Assistant Reference*). They are indepent of the Core, therefore any software distribution can carry its own assistants and drop them into the directory from where DevAssistant loads them - they will be loaded on next invocation. Note, that there is also a possibility to write assistants in Python, but this is no longer supported and will be removed in near future.

1.2.8 Assistants

Internally, each assistant is represented by instance of `devassistant.yaml_assistant.YamlAssistant`. Instances are constructed by DevAssistant in runtime from parsed yaml files. Each assistant can have zero or more subassistants. This effectively forms a tree-like structure. For example:

```
       MainAssistant
       /           \
      /             \
   Python           Ruby
   /   \             / \
  /     \           /   \
Django  Flask    Rails Sinatra
```

This structure is defined by filesystem hierarchy as explained in *Assistants Loading Mechanism*

Each assistant can optionally define arguments that it accepts (either on commandline, or from GUI). For example, you can run the leftmost path with:

```
$ da crt python [python assistant arguments] django [django assistant arguments]
```

If an assistant has any subassistants, one of them **must** be used. E.g. in the example above, you can't use just Python assistant, you have to choose between Django and Flask. If Django would get a subassistant, it wouldn't be usable on its own any more, etc.

Assistant Roles

The `crt` in the above example means, that we're running an assistant that creates a project.

There are four assistant roles:

creator (`crt` in short) creates new projects

modifier (`mod` in short) modifies existing projects

preparer (`prep` in short) prepares environment for development of upstream projects

task (`task` in short) performs arbitrary tasks not related to a specific project

The main purpose of having roles is separating different types of tasks. It would be confusing to have e.g. `python django` assistant (that creates new project) side-by-side with `eclipse` assistant (that registers existing project into Eclipse).

1.2.9 Contributing

If you want to contribute (bug reporting, new assistants, patches for core, improving documentation, ...), please use our Github repo:

- code: https://github.com/bkabrda/devassistant
- issue tracker: https://github.com/bkabrda/devassistant/issues

If you have DevAssistant installed (version 0.8.0 or newer), there is a fair chance that you have `devassistant` preparer. Just run `da prep devassistant` and it will checkout our sources and do all the boring stuff that you'd have to do without DevAssistant.

If you don't have DevAssistant installed, you can checkout the sources like this (just copy&paste this to get the job done):

```
git clone https://github.com/bkabrda/devassistant
# get the official set of assistants
cd devassistant
git submodule init
git submodule update
```

You can find list of core Python dependencies in file `requirements.txt`. If you want to write and run tests (you should!), install dependencies from `requirements-devel.txt`:

```
pip install -r requirements-devel.txt
```

On top of that, you'll need `polkit` for requesting root privileges for dependency installation etc. If you want to play around with GUI, you have to install `pygobject`, too (see how hard this is compared to `da prep devassistant`?)

Overview

This is documentation for version 0.8.0.

DevAssistant is developer's best friend (right after coffee).

DevAssistant can help you with creating and setting up basic projects in various languages, installing dependencies, setting up environment etc. There are three main types of functionality provided:

- `da crt` - create new project from scratch
- `da mod` - take local project and do something with it (e.g. import it to Eclipse)
- `da prep` - prepare development environment for an upstream project or a custom task

DevAssistant is based on idea of per-{language/framework/...} "assistants" with hierarchical structure. E.g. you can run:

```
$ da crt python django -n ~/myproject # sets up Django project named "myproject" inside your home di
$ da crt python flask -n ~/flaskproject # sets up Flask project named "flaskproject" inside your home
$ da crt ruby rails -n ~/alsomyproject # sets up RoR project named "alsomyproject" inside your home d
```

DevAssistant also allows you to work with a previously created project, for example import it to Eclipse:

```
$ da mod eclipse # run in project dir or use -p to specify path
```

With DevAssistant, you can also prepare environment for developing upstream projects - either using project-specific assistants or using "custom" assistant for arbitrary projects (even those not created by DevAssistant):

```
$ da prep custom custom -u scm_url
```

Warning: The `custom` assistant executes custom pieces of code from `.devassistant` file of the project. Therefore you have to be extra careful and use this **only with projects whose authors you trust**.

Last but not least, DevAssistant allows you to perform arbitrary tasks not related to a specific project:

DevAssistant works on Python 2.6, 2.7 and >= 3.3.

This whole project is licensed under GPLv2+.