DevAssistant Documentation

Release 0.7.0

Bohuslav Kabrda, Petr Hracek

October 02, 2013

CONTENTS

DevAssistant - start developing with ease.

CHAPTER

ONE

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
```

Last but not least, DevAssistant allows you to prepare environment for executing arbitrary tasks or developing upstream projects (either using "custom" assistant for projects previously created by DevAssistant or using specific assistant for specific projects):

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

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 three assistant roles:

creator creates new projects

modifier modifies existing projects

preparer prepares environment for development of upstream project or custom task

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 executing aribtrary tasks or development of already existing project located in a remote SCM (e.g. Github). For custom projects created by DevAssistant, you can use the `custom` assistant:

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

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...

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

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 Path

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/`

Each load path entry has this structure:

```
assistants/
  crt/
  mod/
  prep/
files/
  crt/
  mod/
  prep/
  snippets/
icons/
  crt/
  mod/
  prep/
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`) and Preparer (`prep`) 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).

1.2.2 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

There are three types of assistants:

Creator creator assistants are meant to create new projects from scratch, they're accessed using `da` binary

Modifier modifier assistants are used for modifying existing projects previously created by DevAssistant

Preparer preparer assistants are used for setting up environment for already existing projects (located e.g. at remote SCM etc.) that may or may not have been creating by DevAssistant

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* and *Preparer Assistants* that talk about specifics of Modifier, resp. Preparer 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`

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"`)

call installs dependencies from snippet or other dependency section of this assistant. For example:

```
dependencies:
- call: foo # will install dependencies from snippet "foo", section "dependencies"
- call: foo.dependencies_bar # will install dependencies from snippet "foo", section "bar"
- call: self.dependencies_baz # will install dependencies from section "dependencies_baz" of thi
```

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:
  - call: spam.spamspam
- rpm: [ham]
```

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)

snippet 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. By default, section named `run` is invoked (this is a bit different for Modifier Assistants). If there is a section named `run_foo` and `foo` argument is used, then **only** `run_foo` is invoked. This is different from dependencies sections, as the default `dependencies` section is used every time.

Every `run` section is a sequence of various commands, mostly invocations of commandline. Each command is a mapping `command_type: command`. 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 can 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 follows:

cl runs given command on commandline, aborts execution of the invoked assistant if it fails. **Note:** `cd` is a special cased command, which doesn't do shell expansion other than user home dir (~) expansion.

cl_i the `i` option makes the command execution be logged at `INFO` level (default is `DEBUG`), therefore visible to user

log_[diwec] logs given message at level specified by the last letter in `log_X`. If the level is `e` or `c`, the execution of the assistant is interrupted immediately.

dda_{c,dependencies,run}

- `c` creates `.devassistant` file (containing some sane initial meta information about the project) in given directory
- `dda_dependencies` let's you install dependencies from `.devassistant` file (DevAssistant will use dependencies from original assistant and specified `dependencies` attribute, if any - this has the same structure as `dependencies` in normal assistants, and is evaluated in current assistant context, not the original assistant context)
- `dda_run` will execute a series of commands from `run` section from `.devassistant` (in context of current assistant)

if <expression>,else conditional execution. The condition must be an Expression.

for <var> in <expression> (for example `for $i in $(ls)`) - loop over *result* of given expression (if it is string, which almost always is, it is split on whitespaces)

$foo assigns *result* of an Expression to the given variable (doesn't interrupt the assistant execution if command fails)

$success, $foo assigns *logical result* (`True`/`False`) of evaluating an Expression to `$success` and result to `$foo` (same as above)

call run another section of this assistant (e.g.''call: self.run_foo'') of a snippet run section (`call: snippet_name.run_foo`) at this place and then continue execution

scl run a whole section in SCL environment of one or more SCLs (note: you **must** use the scriptlet name - usually `enable` - because it might vary) - for example:

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

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.

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 expressions, really. They 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`/`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`
- `not` - negates the *logical result* of an expression, while leaving *result* intact, can only be used once (no, you can't use `not not not $foo`, sorry)
- `defined $foo` - works exactly as `$foo`, but has *logical result* `True` even if the value is empty or `False`

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 read the whole .devassistant file and make its contents available as any other variables (notably `$subassistant_path`).
- They use dependency sections according to the normal rules + they use *all* the sections that are named according to loaded `$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 set up environment for executing arbitrary tasks or prepare environment and checkout existing upstream projects (possibly using their `.devassistant` file, if they have it). 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.

1.2.3 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.

1.2.4 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.

1.2.5 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]
```

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

There are three assistant roles:

creator creates new projects

modifier modifies existing projects

preparer prepares environment for development of upstream project or custom task

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).

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.

1.2.6 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

Unless you actually 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`. On top of that, you'll need `pygobject` if you want to play around with GUI. DevAssistant also assumes that `git` is installed on your system.

In next version, we will include a `prep` assistant, that will be able to actually do this for you... Sweet, ain't it?

CHAPTER

TWO

OVERVIEW

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
```

Last but not least, DevAssistant allows you to prepare environment for executing arbitrary tasks or developing upstream projects (either using "custom" assistant for projects previously created by DevAssistant or using specific assistant for specific projects):

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

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

This whole project is licensed under GPLv2+.