appi Documentation

Release 0.0.2

Antoine Pinsard

Aug 20, 2017

Contents

CHAPTER 1

Install appi

Installation from portage tree

First check if `dev-python/appi` is already in your portage tree:

```
emerge -av dev-python/appi
```

Installation from sapher overlay

If your distribution does not provide a `dev-python/appi` ebuild, you can get it from the sapher overlay:

```
mkdir -pv /var/overlays
git clone https://github.com/apinsard/sapher-overlay.git /var/overlays/sapher
cat > /etc/portage/repos.conf/sapher <<EOF
[sapher]
location = /var/overlays/sapher
sync-type = git
sync-uri = git://github.com/apinsard/sapher-overlay.git
auto-sync = yes
EOF
emerge -av dev-python/appi::sapher
```

Installation from pypi

Not yet available.

Installation from git repository

```
pip install git+ssh://git@github.com/apinsard/appi.git
```

CHAPTER 2

Get Started

Let's start python3, and write some `appi` calls:

```
$ python3
Python 3.4.5 (default, Nov 29 2016, 00:11:56)
[GCC 5.3.0] on linux
type "help", "copyright", "credits" or "license" for more information.
>>> import appi
>>>
```

Play with atoms

Something you must be familiar with are "query atoms", these are the strings used to query atoms with `emerge` and such tools. `appi.QueryAtom` is a class representing this kind of atoms, it enables you to check if a string is a valid atom or not.

Note: There is also a `DependAtom` which represents a dependency atom as found in ebuilds. It is not covered in this quick start but it behaves, to some extent, the same as `QueryAtom`.

Check atom validity

```
>>> appi.QueryAtom('dev-python/appi')
<QueryAtom: 'dev-python/appi'>
>>> appi.QueryAtom('=sys-apps/portage-2.4.3-r1')
<QueryAtom: '=sys-apps/portage-2.4.3-r1'>
>>> appi.QueryAtom('This is not a valid atom')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python3.4/site-packages/appi/atom.py", line 62, in __init__
    raise AtomError("{atom} is not a valid atom.", atom_string)
```

```
appi.atom.AtomError: This is not a valid atom is not a valid atom.
>>> from appi.exception import AtomError
>>> try:
...     appi.QueryAtom('>=dev-lang/python')
... except AtomError:
...     False
... else:
...     True
...
False
>>>
```

Note: `QueryAtom` only checks that the atom string is **valid**, not that an ebuild actually exists for this atom.

```
>>> appi.QueryAtom('this-package/does-not-exist')
<QueryAtom: 'this-package/does-not-exist'>
>>> appi.QueryAtom('~foo/bar-4.2.1')
<QueryAtom: '~foo/bar-4.2.1'>
>>>
```

Note: If you try to parse atoms without category name, you will notice that it raises an `AtomError` while it is actually a valid atom. There is a `strict` mode enabled by defaut, which makes package category mandatory in order to avoid dealing with ambiguous packages. You can easily disable this behavior by setting `strict=False`.

```
>>> appi.QueryAtom('=portage-2.4.3-r1')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python3.4/site-packages/appi/atom.py", line 71, in __init__
    atom_string, code='missing_category')
appi.atom.AtomError: =portage-2.4.3-r1 may be ambiguous, please specify the category.
>>> appi.QueryAtom('=portage-2.4.3-r1', strict=False)
<QueryAtom: '=portage-2.4.3-r1'>
>>>
```

Inspect atom parts

`QueryAtom` does not only check atoms validity, it also extracts its components.

```
>>> atom = appi.QueryAtom('=dev-lang/python-3*:3.4::gentoo')
>>> atom
<QueryAtom: '=dev-lang/python-3.4*:3.4::gentoo'>
>>> atom.selector
'='
>>> atom.category
'dev-lang'
>>> atom.package
'python'
>>> atom.version
'3'
>>> atom.postfix
'*'
>>> atom.slot
```

```
'3.4'
>>> atom.repository
'gentoo'
>>> atom2 = appi.QueryAtom('foo-bar/baz')
>>> atom2.selector
>>> atom2.version
>>> atom2.category
'foo-bar'
>>>
```

And much more!

Now, would you like to get the list of ebuilds that satisfy this atom? Nothing's easier!

```
>>> atom = appi.QueryAtom('=dev-lang/python-3*:3.4::gentoo')
>>> atom.list_matching_ebuilds()
{<Ebuild: 'dev-lang/python-3.5.2::gentoo'>, <Ebuild: 'dev-lang/python-3.4.3-r1::gentoo
↪'>, <Ebuild: 'dev-lang/python-3.4.5::gentoo'>}
>>>
```

Warning: Yes, this returns python 3.5.2! This version of `appi` is still experimental and we haven't implemented slot filtering yet.

This feature is planned in version `0.1`. See issue #2 if you would like to be informed on progress, or if you want to get involved and help us implement it.

Well, this brings us to ebuilds.

Go on with ebuilds

An `appi.Ebuild` instance represents the file describing a given version of a given package.

Check ebuild validity

Just as with atoms, you can check the validity of an ebuild by instantiating it.

```
>>> appi.Ebuild('/usr/portage/sys-devel/clang/clang-9999.ebuild')
<Ebuild: 'sys-devel/clang-9999::gentoo'>
>>> appi.Ebuild('/home/tony/Workspace/Funtoo/sapher-overlay/x11-wm/qtile/qtile-0.10.6.
↪ebuild')
<Ebuild: 'x11-wm/qtile-0.10.6::sapher'>
>>> appi.Ebuild('/usr/portage/sys-devel/clang/9999.ebuild')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python3.4/site-packages/appi/ebuild.py", line 58, in __init__
    raise EbuildError("{ebuild} is not a valid ebuild path.", path)
appi.ebuild.EbuildError: /usr/portage/sys-devel/clang/9999.ebuild is not a valid␣
↪ebuild path.
>>> from appi.exception import EbuildError
>>> try:
...     appi.Ebuild('/usr/portage/sys-devel/clang/clang-9999')
```

```
... except EbuildError:
...     False
... else:
...     True
...
False
>>> appi.Ebuild('/Unexisting/overlay/path/foo/bar/bar-1.5a_pre5-r12.ebuild')
<Ebuild: 'foo/bar-1.5a_pre5-r12'>
>>>
```

Warning: Note that currently, valid paths to unexisting files are considered valid ebuilds. This behavior is very likely to change as of version `0.1` since reading the ebuild file will be needed to extract some information such as slots and useflags. Thus, the `Ebuild` constructor may also raise `OSError` exceptions such as `FileNotFoundError` in future versions.

Inspect ebuild parts

```
>>> e = appi.Ebuild('/usr/portage/sci-libs/gdal/gdal-2.0.2-r2.ebuild')
>>> e.category
'sci-libs'
>>> e.package
'gdal'
>>> e.version
'2.0.2-r2'
>>> e.repository
<Repository: 'gentoo'>
>>> e.repository.location
PosixPath('/usr/portage')
>>>
```

And much more

You can check if an ebuild matches a given atom:

```
>>> e = appi.Ebuild('/usr/portage/app-portage/gentoolkit/gentoolkit-0.3.2-r1.ebuild')
>>> e.matches_atom(appi.QueryAtom('~app-portage/gentoolkit-0.3.2'))
True
>>> e.matches_atom(appi.QueryAtom('>gentoolkit-0.3.2', strict=False))
True
>>> e.matches_atom(appi.QueryAtom('>=app-portage/gentoolkit-1.2.3'))
False
>>> e.matches_atom(appi.QueryAtom('=app-portage/chuse-0.3.2-r1'))
False
>>>
```

Finally, let's checkout versions

Atom and ebuild objects both define a `get_version()` method that returns the version number as a `Version` object.

```
>>> atom = appi.QueryAtom('=x11-wm/qtile-0.10*')
>>> atom.version
'0.10'
>>> atom.get_version()
<Version: '0.10'>
>>> [(eb, eb.get_version()) for eb in atom.list_matching_ebuilds()]
[(<Ebuild: 'x11-wm/qtile-0.10.5::gentoo'>, <Version: '0.10.5'>), (<Ebuild: 'x11-wm/
↪qtile-0.10.6::gentoo'>, <Version: '0.10.6'>)]
>>>
```

Inspect version parts

```
>>> v = Version('3.14a_beta05_p4_alpha11-r16')
>>> v.base
'3.14'
>>> v.letter
'a'
>>> v.suffix
'_beta05_p4_alpha11'
>>> v.revision
'16'
>>>
```

Compare versions

```
>>> v1 = Version('2.76_alpha1_beta2_pre3_rc4_p5')
>>> v2 = Version('1999.05.05')
>>> v3 = Version('2-r5')
>>> v1 == v2
False
>>> v1 > v3
True
>>> v1 < v2
True
>>> v3.startswith(v1)
False
>>> Version('0.0a-r1').startswith(Version('0.0'))
True
>>>
```

CHAPTER 3

Reference

appi

appi.conf

appi.conf.Repository

appi.exception

appi.exception.AppiError

appi.exception.AtomError

appi.exception.EbuildError

appi.exception.PortageError

appi.exception.VersionError

appi.DependAtom

appi.Ebuild

Ebuild(path)

Create an ebuild object from an absolute path. `path` must be a valid ebuild path. A valid ebuild path starts with the repository location, then a category directory, a package directory and a package/version file with `.ebuild` extension.

Raises

- *EbuildError* if `path` is not a valid ebuild path.

Examples

```
>>> appi.Ebuild('/usr/portage/x11-wm/qtile/qtile-0.10.6.ebuild')
<Ebuild 'x11-wm/qtile-0.10.6::gentoo'>
>>> appi.Ebuild('/home/tony/Workspace/Funtoo/sapher-overlay/x11-wm/qtile/qtile-0.10.6.
↪ebuild')
<Ebuild 'x11-wm/qtile-0.10.6::sapher'>
>>> appi.Ebuild('/undefined/x11-wm/qtile/qtile-0.10.6.ebuild')
<Ebuild 'x11-wm/qtile-0.10.6'>
>>> appi.Ebuild('/x11-wm/qtile/qtile-0.10.6.ebuild')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.4/site-packages/appi/ebuild.py", line 59, in __init__
    raise EbuildError("{ebuild} is not a valid ebuild path.", path)
appi.ebuild.EbuildError: /x11-wm/qtile/qtile-0.10.6.ebuild is not a valid ebuild path.
>>>
```

Attributes

- **category** (`str`) The package category
- **package** (`str`) The package name
- **version** (`str`) The package version
- **repository** (*appi.conf.Repository*) The package repository if available, `None` otherwise

Examples

```
>>> e = appi.Ebuild('/usr/portage/www-client/brave/brave-0.12.15.ebuild')
>>> e.category
'www-client'
>>> e.package
'brave'
>>> e.version
'0.12.15'
>>> e.repository
<Repository 'gentoo'>
>>> f = appi.Ebuild('/tmp/www-client/brave/brave-0.12.15.ebuild')
>>> f.repository
>>>
```

String representation

The string representation of an ebuild is as following: `<category>/<name>-<version>`. Also, if the repository is known, it is appended as `::<repository>`.

Examples

```
>>> str(appi.Ebuild('/usr/portage/dev-python/appi/appi-0.0.ebuild'))
'dev-python/appi-0.0::gentoo'
>>> str(appi.Ebuild('/home/tony/Workspace/Funtoo/sapher-overlay/dev-python/appi/appi-
↪1.0.ebuild'))
'dev-python/appi-1.0::sapher'
>>> str(appi.Ebuild('/not/a/repository/dev-python/appi/appi-0.1.ebuild')
'dev-python/appi-0.1'
>>>
```

get_version() -> appi.Version

`Ebuild.version` is a string representing the version of the ebuild. `get_version()` returns it as a *Version* object.

Examples

```
>>> e = appi.Ebuild('/usr/portage/media-libs/libcaca/libcaca-0.99_beta19.ebuild')
>>> e.version
'0.99_beta19'
>>> e.get_version()
<Version '0.99_beta19'>
```

matches_atom(atom) -> bool

Return `True` if the ebuild matches the given `atom`.

Warning: This method still lacks SLOT check. It should be implemented in version `0.1`.

Examples

```
>>> e = appi.Ebuild('/usr/portage/media-gfx/blender/blender-2.72b-r4.ebuild')
>>> e.matches_atom(QueryAtom('=media-gfx/blender-2.72b-r4'))
True
>>> e.matches_atom(QueryAtom('media-gfx/gimp'))
False
>>> e.matches_atom(QueryAtom('~media-gfx/blender-2.72b'))
True
>>> e.matches_atom(QueryAtom('>media-gfx/blender-2.72'))
True
>>> e.matches_atom(QueryAtom('<=media-gfx/blender-2.72'))
False
>>> e.matches_atom(QueryAtom('=media-gfx/blender-2*'))
True
>>>
```

appi.QueryAtom

A "query atom" is an atom that is used for querying a package, this is the kind of atoms accepted by `emerge` for instance.

There also exist *DependAtom* that have a slightly different format and is used by `DEPEND` variables in ebuilds.

QueryAtom(atom_string, strict=True)

Create a query atom from its string representation. `atom_string` must be a valid string representation of a query atom. The `strict` argument controls the "strict mode" state. When strict mode is enabled, the package category is mandatory and an error will be raised if it is missing. When strict mode is disabled, the package category is optional, which makes, for instance, `=firefox-50-r1` a valid atom.

Raises

- *AtomError* if `atom_string` is not a valid atom taking `strict` mode into consideration. Possible error codes:
  - **missing_category** The package category is missing, this can be ignored by setting `strict=False`.
  - **missing_selector** The package version was specified but the version selector is missing.
  - **missing_version** The version selector was specified but the package version is missing.
  - **unexpected_revision** The version contains a revision number while the version selector is ~.
  - **unexpected_postfix** The * postfix is specified but the version selector is not =.

Examples

```
>>> appi.QueryAtom('>=www-client/firefox-51')
<QueryAtom: '>=www-client/firefox-51'>
>>> appi.QueryAtom('=www-client/chromium-57*')
<QueryAtom: '=www-client/chromium-57*'>
>>> appi.QueryAtom('www-client/lynx')
<QueryAtom: 'www-client/lynx'>
>>> appi.QueryAtom('=www-client/links')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python3.5/site-packages/appi/atom.py", line 79, in __init__
    code='missing_version')
appi.atom.AtomError: =www-client/links misses a version number.
>>> appi.QueryAtom('google-chrome')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python3.5/site-packages/appi/atom.py", line 71, in __init__
    atom_string, code='missing_category')
appi.atom.AtomError: google-chrome may be ambiguous, please specify the category.
>>> appi.QueryAtom('google-chrome', strict=False)
<QueryAtom: 'google-chrome'>
```

Attributes

- **selector** (`str`) The package selector (>=, <=, <, =, > or ~)

- **category** (`str`) The package category
- **package** (`str`) The package name
- **version** (`str`) The package version
- **postfix** (`str`) The package postfix (`*` is the only possible value)
- **slot** (`str`) The package slot
- **repository** (`str`) The name of the repository

All these attribute, excepted **package**, are optional and may be `None`.

Examples

```
>>> a = appi.QueryAtom('=dev-db/mysql-5.6*:5.6::gentoo')
>>> a.selector
'='
>>> a.category
'dev-db'
>>> a.package
'mysql'
>>> a.version
'5.6'
>>> a.postfix
'*'
>>> a.slot
'5.6'
>>> a.repository
'gentoo'
>>> b = appi.QueryAtom('~postgresql-9.4', strict=False)
>>> b.selector
'~'
>>> b.category
>>> b.package
'postgresql'
>>> b.version
'9.4'
>>> b.postfix
>>> b.slot
>>> b.repository
>>>
```

String Representation

The string representation of an atom is the raw atom string itself: `<selector><category>/<package>-<version><postfix>:<slot>::<repository>`

Examples

```
>> str(appi.QueryAtom('dev-db/postgresql'))
'dev-db/postgresql'
>>> str(appi.QueryAtom('<dev-db/postgresql-9.6'))
'<dev-db/postgresql-9.6'
>>> str(appi.QueryAtom('>=dev-db/postgresql-8.4-r1::gentoo'))
```

```
'>=dev-db/postgresql-8.4-r1::gentoo'
>>> str(appi.QueryAtom('dev-db/postgresql:9.4'))
'dev-db/postgresql:9.4'
>>> a = appi.QueryAtom('=postgresql-9.4-r1', strict=False)
>>> str(a)
'=postgresql-9.4-r1'
>>> a.category = 'dev-db'
>>> str(a)
'=dev-db/postgresql-9.4-r1'
```

Warning: This can be useful to change the package category of an existing instance as above if you want to read atoms without requiring category and infer it afterwards if it is not a ambiguous.

However, it is not recommended to change other attributes values. Validity won't be checked and this can lead to incoherent atoms as illustrated below. We don't prevent attributes from being altered, we assume you are a sane minded developer who knows what he is doing.

```
>>> # /!\ DONT DO THIS /!\
>>> a.selector = ''
>>> str(a)
'dev-db/postgresql-9.4-r1'
>>> # Why would you anyway?
>>>
```

get_version() -> appi.Version

`QueryAtom.version` is a string representing the version included in the atom. `get_version()` returns it as a *Version* object.

Examples

```
>>> a = appi.QueryAtom('>=media-gfx/image-magick-7.0:0/7.0.4.3')
>>> a.version
'7.0'
>>> a.get_version()
<Version '7.0'>
```

get_repository() -> appi.conf.Repository

`QueryAtom.repository` is the name of the repository included in the atom. `get_repository()` returns the repository a *Repository* object.

This may be useful if you want to get the path or other data from the repository.

Examples

```
>>> a = appi.QueryAtom('app-portage/chuse::gentoo')
>>> a.repository
'gentoo'
```

```
>>> a.get_repository()
<Repository: 'gentoo'>
>>> a = appi.QueryAtom('app-portage/chuse::sapher')
>>> appi.QueryAtom('app-portage/chuse::sapher').get_repository()
<Repository: 'sapher'>
>>> appi.QueryAtom('app-portage/chuse::unexisting').get_repository()
>>>
```

list_matching_ebuilds() -> {appi.Ebuild, ...}

Returns the `set` of all ebuilds matching this atom.

Examples

```
>>> appi.QueryAtom('app-portage/chuse').list_matching_ebuilds()
{<Ebuild: 'app-portage/chuse-1.0.2::gentoo'>, <Ebuild: 'app-portage/chuse-1.1::gentoo
↪'>,
<Ebuild: 'app-portage/chuse-1.0.2::sapher'>, <Ebuild: 'app-portage/chuse-1.1::sapher'>
↪}
>>> appi.QueryAtom('app-portage/chuse::gentoo').list_matching_ebuilds()
{<Ebuild: 'app-portage/chuse-1.0.2::gentoo'>, <Ebuild: 'app-portage/chuse-1.1::gentoo
↪'>}
>>> appi.QueryAtom('screen', strict=False).list_matching_ebuilds()
{<Ebuild: 'app-misc/screen-4.0.3-r9::funtoo-overlay'>, <Ebuild: 'app-vim/screen-1.
↪5::gentoo'>,
<Ebuild: 'app-misc/screen-4.0.3-r9::gentoo'>, <Ebuild: 'app-misc/screen-4.4.0::funtoo-
↪overlay'>,
<Ebuild: 'app-misc/screen-4.0.3-r10::funtoo-overlay'>,
<Ebuild: 'app-misc/screen-4.2.1-r2::funtoo-overlay'>, <Ebuild: 'app-misc/screen-4.4.
↪0::gentoo'>,
<Ebuild: 'app-misc/screen-4.0.3-r10::gentoo'>, <Ebuild: 'app-misc/screen-4.0.3-
↪r3::gentoo'>,
<Ebuild: 'app-misc/screen-4.0.3-r3::funtoo-overlay'>, <Ebuild: 'app-misc/screen-4.2.1-
↪r2::gentoo'>}
>>> appi.QueryAtom('<screen-4', strict=False).list_matching_ebuilds()
{<Ebuild: 'app-vim/screen-1.5::gentoo'>}
>>> appi.QueryAtom('<screen-1', strict=False).list_matching_ebuilds()
set()
>>> appi.QueryAtom('dev-lang/python:3.4::gentoo').list_matching_ebuilds()
{<Ebuild: 'dev-lang/python-3.4.5::gentoo'>, <Ebuild: 'dev-lang/python-3.6.0::gentoo'>,
<Ebuild: 'dev-lang/python-3.5.2::gentoo'>, <Ebuild: 'dev-lang/python-2.7.12::gentoo'>}
>>>
```

Warning: As you can see, the last example illustrates that slots are not yet taken into account in ebuilds filtering. It is scheduled in version `0.1`. See issue #2 if you would like to be informed on progress, or if you want to get involved and help us implement it.

matches_existing_ebuild() -> bool

Returns `True` if any existing ebuild matches this atom. `False` otherwise. Basically, it checks if `list_matching_ebuilds()` returns an empty set or not.

Examples

```
>>> appi.QueryAtom('dev-python/unexisting-module').matches_existing_ebuild()
False
>>> appi.QueryAtom('dev-python/appi').matches_existing_ebuild()
True
>>> appi.QueryAtom('~dev-python/appi-1.2.3').matches_existing_ebuild()
False
>>> appi.QueryAtom('screen', strict=False).matches_existing_ebuild()
True
>>>
```

appi.Version

The `Version` object is the representation of a package version. It enables to compare versions.

Version(version_string)

Create a version object from a valid version string.

Raises

- *VersionError* if `version_string` is not a valid version number

Examples

```
>>> appi.Version('1.3')
<Version: '1.3'>
>>> appi.Version('3.14-r1')
<Version: '3.14-r1'>
>>> appi.Version('1.2.3a_rc4_pre5_alpha2-r6')
<Version: '1.2.3a_rc4_pre5_alpha2-r6'>
>>> appi.Version('2.0beta')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python3.5/site-packages/appi/version.py", line 76, in __init__
    "{version} is not a valid version.", version_string)
appi.version.VersionError: 2.0beta is not a valid version.
>>> appi.Version('2.0_beta')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python3.5/site-packages/appi/version.py", line 76, in __init__
    "{version} is not a valid version.", version_string)
appi.version.VersionError: 2.0_beta is not a valid version.
>>> appi.Version('bonjour')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python3.5/site-packages/appi/version.py", line 76, in __init__
    "{version} is not a valid version.", version_string)
appi.version.VersionError: bonjour is not a valid version.
>>>
```

Attributes

- **base** (`str`) The base version number (part before the letter if any)
- **letter** (`str`) The letter version number (a single letter), optional
- **suffix** (`str`) The suffix version number (release, pre-release, patch, ...), optional
- **revision** (`str`) The ebuild revision number, optional

Examples

```
>>> v = Version('1.2.3d_rc5_p0-r6')
>>> v.base
'1.2.3'
>>> v.letter
'd'
>>> v.suffix
'_rc5_p0'
>>> v.revision
'6'
>>>
```

compare(other) -> `int`

startswith(version) -> `bool`

get_upstream_version() -> `Version`