Python RPM Porting Guide

Release 0.1

Red Hat, Inc.

Apr 27, 2017

Contents

The section for applications is for software where the user doesn't care in which programming language it is written. For example, it doesn't matter if it is written in Python 2 or 3 because the application should run the same.

Applications have to have an executable, which you can check by running `dnf repoquery -l your-package-name | grep /usr/bin/`.

If your package is not being imported by third-party projects (e.g. `import some_module` in a Python file), it is most likely an application.

Try running `dnf repoquery --whatrequires your-package-name` to see a list of packages that depend on yours. If there are none, your package is likely an application, because there would be little reason to package a module that nobody uses. However, if there are some packages that depend on yours, we cannot be sure if it's an application-only package or not, as some of these packages might be depending on your application itself, instead of importing modules from your package.

If this doesn't fit your package, look into *other sections*.

Table of Contents

Porting the specfile to Python 3

Applications behave the same when run under Python 2 and Python 3, therefore all you need to do is change the spec file to use Python 3 instead of Python 2.

In essence, porting of an application RPM mostly consists of going through the spec file and adding number 3 or substituting number 3 for number 2 where appropriate. Occasionally also substituting old macros for new ones that are more versatile.

So let's take an example spec file and port it to illustrate the process. We start with a spec file for an application that is being run with Python 2:

```
%global srcname example

Name:           %{srcname}
Version:        1.2.3
Release:        1%{?dist}
Summary:        An example Python application

License:        MIT
URL:            http://pypi.python.org/pypi/%{srcname}
Source0:        https://files.pythonhosted.org/packages/source/e/%{srcname}/%{srcname}
↪-%{version}.tar.gz

BuildArch:      noarch
BuildRequires:  python-devel

%description
A Python application which provides a convenient example.


%prep
%autosetup -n %{srcname}-%{version}


%build
%{__python} setup.py build
```

```
%install
%{__python} setup.py install --skip-build --root $RPM_BUILD_ROOT


%check
%{__python} setup.py test


%files
%license COPYING
%doc README
%{python_sitelib}/*
%{_bindir}/sample-exec


%changelog
...
```

Modifications

First it is recommended to update the software you are packaging to its newest upstream version. If it already is at the latest version, increment the release number. Don't forget to add a `%changelog` entry as well.

Note: In this document you will encounter lot of RPM macros. You can look up many of the Python macros in the Python packaging guidelines (click the **Expand button** on the right side).

BuildRequires and Requires

Change `BuildRequires` from `python-devel` to `python3-devel` and adjust all other `Requires` and `BuildRequires` entries to point only to python3 packages.

It is very important that you don't use any Python 2 dependencies as that would make your package depend both on Python version 2 and version 3, which would render your porting efforts useless.

%build

In the build section you can find a variety of macros, for example `%py_build` and its newer version `%py2_build`. You can freely substitute these by the new Python 3 macro `%py3_build`.

Occasionally you will find a custom build command prefixed by the `%{__python}` or `%{__python2}` macros, or in some cases just prefixed by the Python interpreter invoked without a macro at all, e.g.:

```
%{__python} setup.py build
    or
python setup.py build
```

In these cases first try substituting the whole build command by the new smart macro `%py3_build` which should in many cases correctly figure out what ought to be done automatically. Otherwise change the invocation of the Python interpreter to the `%{__python3}` macro.

In rare cases, you might encounter some non-Python build script such as a Makefile. In these instances you have to adjust the script on your own, consult the documentation for the specific build method.

%install and %check

The `%install` section will oftentimes contain the `%py_install` and `%py2_install` macros; you can replace these with the new Python 3 macro `%py3_install`.

Furthermore, as in the preceding *%build* section, you will frequently find custom scripts or commands prefixed by either the `%{__python}` or `%{__python2}` macros or simply preceded by an invocation of the Python interpreter without the use of macros at all.

In the install section, try substituting it with the new `%py3_install` macro, which should figure out what to do automatically. If that doesn't work, or if you're porting the `%check` section, just make sure that any custom scripts or commands are invoked by the new `%{__python3}` macro.

Again as in the *%build* section, in the rare cases where you encounter a non-Python install script such as a Makefile, consult documentation for the specific install method and make adjustments on your own.

Lastly, in the `%check` section you can also encounter a custom Python command that runs the tests, such as `nosetests` or `py.test`. In that case find out what is the name of the executable for Python 3 and use it instead of the Python 2 version.

```
%check
py.test-3
or
nosetests-%{python3_version}
```

As you can see on the example of the `nosetests`, not all packages follow the proper naming conventions for executables. To list what executables a package contains, you can use:

```
$ dnf repoquery -l python3-nose | grep /usr/bin/
/usr/bin/nosetests-3.4
```

%files

In the files section you can regularly find the following macros: `%{python2_sitelib}`, `%{python2_sitearch}`, `%{python2_version}`, `%{python2_version_nodots}` or their unversioned alternatives. Substitute these with their counterparts for Python 3, e.g. `%{python3_sitelib}`.

The files section may also contain the versioned executable, usually `%{_bindir}/sample-exec-2.7` in which case it should be substituted by `%{_bindir}/sample-exec-%{python3_version}`.

Are shebangs dragging you down (to Python 2)?

A shebang is an indicator on the first line of an executable script that indicates in what interpreter is the script supposed to be launched, examples include *python, bash and perl*. When software gets ported to Python 3, the lone shebang often remains forgotten and keeps pointing to Python 2. In most cases this is handled automatically: the `setup.py` script (usually run through the `%py3_build` and `%py3_install` RPM macros) adjusts shebangs for you. However, sometimes it's up to you to handle the situation.

RPM has very good capabilities of automatically finding dependencies, and one of the ways it accomplishes that is by looking at the shebangs of all the files in the package. Therefore **it is important to check if the shebangs are not dragging in a runtime dependency on Python 2**.

As the porting of the spec file is nearly finished, build it and then run the following analysis on the resulting Python 3 RPM file:

```
$ rpm -qp --requires path/to/an.rpm | grep -E '/usr/bin/(python|env)'
```

This will list all the Python executables your RPM package depends on as well as the `/usr/bin/env` executable which usually invokes `python`. The use of `env` is dangerous: applications should be using the safe system version of Python and not trust whatever version `env` might try to substitute. **If you find that an RPM package for Python 3 depends on Python 2 or `/usr/bin/env` you need to fix it**.

Fixing shebangs

First find out what shebangs are used in your package by unpacking the sources for the project, `cd`-ing into the unpacked directory and trying the following command(s):

```
$ # Searches for all shebangs among the sources
$ grep -r '^#!/' .

$ # Searches only Python shebangs
$ grep -rE '^#!/usr/bin/(python|env python)' .
```

You will usually find one of these two shebangs:

```
#!/usr/bin/python
#!/usr/bin/env python
```

It is advisable to change both of these to `#!/usr/bin/python3`. `/usr/bin/env` can be useful for scripts, but applications should link to the system version of Python outright.

To change the shebangs in the files you can use one (or a combination) of the following commands, which you should place at the end of the `%prep` section. They will change the shebangs to point to the Python 3 interpreter stored in the `${__python3}` macro.

```
$ # Change shebang in individual files
$ sed -i '1s=^#!/usr/bin/\(python\|env python\)[0-9.]*=#!%{__python3}=' path/to/file1␣
↪file2 file3 ...

$ # Change shebang in all relevant files in this directory and all subdirectories
$ # See `man find` for how the `-exec command {} +` syntax works
$ find -type f -exec sed -i '1s=^#!/usr/bin/\(python\|env python\)[23]\?=#!%{__
↪python3}=' {} +

$ # Change shebang in all relevant executable files in this directory and all␣
↪subdirectories
$ find -type f -executable -exec sed -i '1s=^#!/usr/bin/\(python\|env python\)[23]\?=
↪#!%{__python3}=' {} +
```

You don't have to worry about accidentally corrupting other files as these scripts will only change a file if the beginning of its first line exactly matches one of the two aforementioned shebangs.

Ported RPM spec file

Here you can peruse the entire ported spec file:

```
%global srcname example

Name:           %{srcname}
```

```
Version:        1.2.3
Release:        2%{?dist}
Summary:        An example Python application

License:        MIT
URL:            http://pypi.python.org/pypi/%{srcname}
Source0:        https://files.pythonhosted.org/packages/source/e/%{srcname}/%{srcname}
↪-%{version}.tar.gz

BuildArch:      noarch
BuildRequires:  python3-devel

%description
A Python application which provides a convenient example.


%prep
%autosetup -n %{srcname}-%{version}


%build
%py3_build


%install
%py3_install


%check
%{__python3} setup.py test


%files
%license COPYING
%doc README
%{python3_sitelib}/*
%{_bindir}/sample-exec


%changelog
...
```

Diff of the changes

And here you can see the diff of the original and the ported spec files to fully observe all the changes that were made:

```
--- specs/application.spec.orig
+++ specs/application.spec
@@ -2,7 +2,7 @@

 Name:           %{srcname}
 Version:        1.2.3
-Release:        1%{?dist}
+Release:        2%{?dist}
 Summary:        An example Python application
```

```
 License:        MIT
@@ -10,7 +10,7 @@
 Source0:        https://files.pythonhosted.org/packages/source/e/%{srcname}/%
↪{srcname}-%{version}.tar.gz

 BuildArch:      noarch
-BuildRequires:  python-devel
+BuildRequires:  python3-devel

 %description
 A Python application which provides a convenient example.
@@ -21,21 +21,21 @@


 %build
-%{__python} setup.py build
+%py3_build


 %install
-%{__python} setup.py install --skip-build --root $RPM_BUILD_ROOT
+%py3_install


 %check
-%{__python} setup.py test
+%{__python3} setup.py test


 %files
 %license COPYING
 %doc README
-%{python_sitelib}/*
+%{python3_sitelib}/*
 %{_bindir}/sample-exec
```

Porting Python modules

This section is for Python **packages that do not have any executables**. Usually, these are normal Python modules that are being imported by third-party projects.

If this doesn't fit your package, look into *other sections*.

Table of Contents

Porting the specfile to Python 3

Because the software you're packaging is going to be imported by third-party projects, it is crucial to think about what Python versions your package will support.

If you switch your package to use only Python 3, suddenly projects running on Python 2 will no longer be able to import your modules. And of course, if you continue using Python 2 only, new Python 3 projects won't get to use your software either.

For these reasons, it is highly advised to **split your package into two subpackages**, one for each major Python version.

Let's take an example spec file and port it to illustrate the process. We start with a spec file for Python module packaged for Python 2.

```
%global srcname example

Name:           python-%{srcname}
Version:        1.2.3
Release:        1%{?dist}
Summary:        An example Python module

License:        MIT
URL:            http://pypi.python.org/pypi/%{srcname}
Source0:        https://files.pythonhosted.org/packages/source/e/%{srcname}/%{srcname}
↪-%{version}.tar.gz

BuildArch:      noarch
BuildRequires:  python-devel
Requires:       python-some-module
Requires:       python2-other-module

%description
A Python module which provides a convenient example.


%prep
%autosetup -n %{srcname}-%{version}


%build
%{__python} setup.py build
```

```
%install
%{__python} setup.py install --skip-build --root $RPM_BUILD_ROOT


%check
%{__python} setup.py test


%files
%license COPYING
%doc README
%{python_sitelib}/*


%changelog
...
```

Modifications

First it is recommended to update the software you are packaging to its newest upstream version. If it already is at the latest version, increment the release number. Don't forget to add a `%changelog` entry as well.

Note: In this document you will encounter lot of RPM macros. You can look up many of the Python macros in the Python packaging guidelines (click the **Expand button** on the right side).

Creating subpackages

Each subpackage you create will need to have its own name, summary and description. If you haven't already, it is thus advised to declare macros for common values at the top of the specfile:

```
%global srcname example
```

Now we can use these to create the subpackages. The following should be placed beneath the `%description` section of the base package:

```
%package -n python2-%{srcname}
Summary:  %{summary}
Requires: python-some-module
Requires: python2-other-module
%{?python_provide:%python_provide python2-%{srcname}}

%description -n python2-%{srcname}
A Python tool which provides a convenient example.


%package -n python3-%{srcname}
Summary:  %{summary}
Requires: python3-some-module
Requires: python3-other-module
%{?python_provide:%python_provide python3-%{srcname}}
```

```
%description -n python3-%{srcname}
A Python tool which provides a convenient example.
```

First, using the `%package` macro you start defining a new *subpackage*, specifying its full name as `python2-%{srcname}`, which in this case will become `python2-example`. Next we provide the summary which we defined earlier. `BuildRequires:` tags from the original spec file will remain where they are—declared in the definition of the base package at the top of the spec file. However, the runtime requirements—the ones listed using the `Requires:` tag—will be different for the two subpackages, so they have to be moved here to the definition of each subpackage.

While you can *cut and paste* all the `Requires:` tags directly from the base package to the `python2-` subpackage, remember that for the `python3-` subpackage you need to find Python 3 versions of each of the runtime dependencies.

Note: You can see that the naming of Python 2 packages isn't uniform: some follow the current convention of using the `python2-` prefix, older ones use only the `python-` prefix, and the oldest might be without a prefix at all.

In many cases the Python 2 package can be found under both the `python2-` and `python-` prefixes, one of them being *virtually provided* by the `Provides:` tag. Whenever possible, use the version with the `python2-` prefix.

%python_provide

Now that we're splitting the package `python-example` into `python2-example` and `python3-example`, we need to define what will happen when the user tries to install the unversioned name `python-example`.

At the time of this writing, the packaging guidelines say that the *default* version should be the one for Python 2. However, it is expected to change to Python 3 some time in the future. To avoid having to adjust all Python packages in Fedora when that time comes, the `%python_provide` macro was devised:

```
%{?python_provide:%python_provide python2-%{srcname}}
and
%{?python_provide:%python_provide python3-%{srcname}}
```

This is a line you should include in each of your subpackages and it works thus: First the part `?python_provide:` checks whether the macro exists and if not, the entire line is ignored. After that we actually use the `%python_provide` macro and give it one argument—the name of the given subpackage.

The macro will then check whether this Python version is default or not—if not, the line is again ignored. However, if indeed this is the currently default Python version, the macro is replaced with a *virtual provides* tag: `Provides: python-%{srcname}`. This will tell the packaging system (dnf, yum, ...) to install this subpackage when user searches for `python-example`.

%description

Each subpackage also needs to contain its own description. However, unlike the `Summary:` and `Requires:` tags, which are automatically applied to the subpackage declared above them, the `%description` macro needs to be told to which subpackage it belongs. You can do that by appending the same name as you did with the `%package` macro itself.

```
%description -n python3-%{srcname}
A Python tool which provides a convenient example.
```

BuildRequires and Requires

Now that you're building subpackages for both Python 2 and Python 3, you need to adjust the `BuildRequires:` by adding Python 3 versions of all the existing build dependencies. Starting with `python-devel`: Use its new version-specific name `python2-devel` and add it's Python 3 equivalent `python3-devel`.

As described *above*, `Requires:` tags are a bit more complicated. You should move the current set of `Requires:` underneath the definition of the Python 2 subpackage, and for the Python 3 subpackage, you need to find Python 3 alternatives for all the current Python 2 runtime requirements that are specified with the `Requires:` tags.

%build

Currently your package is building the software for Python 2, what we need to do is also add building for Python 3. While we're modifying the spec file, however, it's a good idea to also update it to new standards—in this case a new macro.

In the ideal case, you'll find the build done with either the `%py2_build` macro or its older version `%py_build`, which you then should exchange for the former. In either case, you can just add the macro `%py3_build` afterwards, and this part is done. Note that to use these macros, you need to have `python2-devel` and/or `python3-devel` listed among `BuildRequires`, but most Python packages already do.

```
%build
%py2_build
%py3_build
```

In many cases, however, you will find a custom build command prefixed by the `%{__python}` or `%{__python2}` macros, or in some cases just prefixed by the python interpreter invoked without a macro at all, e.g.:

```
%{__python} custombuild.py --many-flags
    or
python custombuild.py --many-flags
```

In these cases first try substituting the whole build command by the new pair of smart macros `%py2_build` and `%py3_build`, which should in many cases correctly figure out what ought to be done automatically. Otherwise, duplicate the entire command and change the invocation of the python interpreter to the `%{__python2}` macro in one of them and to the `%{__python3}` in the other.

```
%build
%{__python2} custombuild.py --many-flags
%{__python3} custombuild.py --many-flags
```

Rarely, you might encounter some non-Python build script such as a Makefile. In these instances you have to adjust the script on your own, consult the documentation for the specific build method.

%install

First, in the same manner as in the preceding *%build* section, it is advisable to upgrade the current Python 2 install command to use the new `%py2_install` macro, however, if that doesn't work for you, you can stick with the current install command, just make sure it's invoked by the `%{__python2}` macro.

After that, add the corresponding Python 3 install command, which will be either the custom command prefixed by `%{__python3}` or the new `%py3_install` macro.

```
%install
%py2_install
%py3_install
```

Again as in the *%build* section, in the rare cases where you encounter a non-Python install script such as a Makefile, consult documentation for the specific install method and make adjustments on your own.

%check

Unlike in previous sections, there's no special macro for the `%check` section, and so here if the original spec file uses any sort of a python script for testing, just make sure that the tests are invoked once using the `%{__python2}` macro and a second time using the `%{__python3}` macro.

```
%check
%{__python2} setup.py test
%{__python3} setup.py test
```

Chances are that you will encounter a custom Python command that runs the tests, such as `nosetests` or `py.test`. In that case find out what is the name of the executable for Python 3 and run it after the Python 2 command.

If the command for Python 2 can be invoked explicitly for Python 2, e.g. as `py.test-2` instead of just `py.test`, use it. Note that to use py.test commands, you need to have `python2-pytest` and/or `python3-pytest` listed among BuildRequires.

```
%check
py.test-2
py.test-3

or

nosetests-%{python2_version}
nosetests-%{python3_version}
```

As you can see on the example of the `nosetests`, not all packages follow the proper naming conventions for executables. To list what executables a package contains, you can use:

```
$ dnf repoquery -l python3-nose | grep /usr/bin/
/usr/bin/nosetests-3.4
```

%files

The presence or absence of a `%files` section is the deciding factor in whether a given package or subpackage gets built or not. Therefore, to assure that our base package doesn't get built (as all the content has been moved to the two subpackages), make sure there is no `%files` section without a subpackage name.

You can reuse the current `%files` section for the Python 2 submodule by giving it the appropriate package name. You can keep it almost the same as before, just make sure that, where appropriate, it uses the new macros `%{python2_sitelib}`, `%{python2_sitearch}`, `%{python2_version}` or perhaps `%{python2_version_nodots}`.

```
%files -n python2-%{srcname}
%license COPYING
%doc README
%{python2_sitelib}/*
```

Accordingly we'll also add a `%files` section for the Python 3 subpackage. You can copy the previous files section, but make sure you change all the Python 2 macros into Python 3 versions.

```
%files -n python3-%{srcname}
%license COPYING
%doc README
%{python3_sitelib}/*
```

Ported RPM spec file

Here you can peruse the entire ported spec file:

```
%global srcname example

Name:           python-%{srcname}
Version:        1.2.3
Release:        2%{?dist}
Summary:        An example Python module

License:        MIT
URL:            http://pypi.python.org/pypi/%{srcname}
Source0:        https://files.pythonhosted.org/packages/source/e/%{srcname}/%{srcname}
↪-%{version}.tar.gz

BuildArch:      noarch
BuildRequires:  python2-devel
BuildRequires:  python3-devel

%description
A Python module which provides a convenient example.


%package -n python2-%{srcname}
Summary:        %{summary}
Requires:       python2-some-module
Requires:       python2-other-module
%{?python_provide:%python_provide python2-%{srcname}}

%description -n python2-%{srcname}
A Python module which provides a convenient example.


%package -n python3-%{srcname}
Summary:        %{summary}
Requires:       python3-some-module
Requires:       python3-other-module
%{?python_provide:%python_provide python3-%{srcname}}

%description -n python3-%{srcname}
A Python module which provides a convenient example.


%prep
%autosetup -n %{srcname}-%{version}


%build
%py2_build
%py3_build
```

```
%install
%py2_install
%py3_install


%check
%{__python2} setup.py test
%{__python3} setup.py test


# Note that there is no %%files section for the unversioned Python package
# if we are building for several Python runtimes
%files -n python2-%{srcname}
%license COPYING
%doc README
%{python2_sitelib}/*

%files -n python3-%{srcname}
%license COPYING
%doc README
%{python3_sitelib}/*


%changelog
...
```

Diff of the changes

And here you can see the diff of the original and the ported spec files to fully observe all the changes that were made:

```
--- specs/module.spec.orig
+++ specs/module.spec
@@ -2,7 +2,7 @@

 Name:           python-%{srcname}
 Version:        1.2.3
-Release:        1%{?dist}
+Release:        2%{?dist}
 Summary:        An example Python module

 License:        MIT
@@ -10,11 +10,30 @@
 Source0:        https://files.pythonhosted.org/packages/source/e/%{srcname}/%
↪{srcname}-%{version}.tar.gz

 BuildArch:      noarch
-BuildRequires:  python-devel
-Requires:       python-some-module
-Requires:       python2-other-module
+BuildRequires:  python2-devel
+BuildRequires:  python3-devel

 %description
+A Python module which provides a convenient example.
```

```
+
+
+%package -n python2-%{srcname}
+Summary:        %{summary}
+Requires:       python2-some-module
+Requires:       python2-other-module
+%{?python_provide:%python_provide python2-%{srcname}}
+
+%description -n python2-%{srcname}
+A Python module which provides a convenient example.
+
+
+%package -n python3-%{srcname}
+Summary:        %{summary}
+Requires:       python3-some-module
+Requires:       python3-other-module
+%{?python_provide:%python_provide python3-%{srcname}}
+
+%description -n python3-%{srcname}
 A Python module which provides a convenient example.


@@ -23,21 +42,31 @@


 %build
-%{__python} setup.py build
+%py2_build
+%py3_build


 %install
-%{__python} setup.py install --skip-build --root $RPM_BUILD_ROOT
+%py2_install
+%py3_install


 %check
-%{__python} setup.py test
+%{__python2} setup.py test
+%{__python3} setup.py test


-%files
+# Note that there is no %%files section for the unversioned Python package
+# if we are building for several Python runtimes
+%files -n python2-%{srcname}
 %license COPYING
 %doc README
-%{python_sitelib}/*
+%{python2_sitelib}/*
+
+%files -n python3-%{srcname}
+%license COPYING
+%doc README
+%{python3_sitelib}/*
```

```
%changelog
```

Porting an application and a module in one package

Use this section if your package contains both a Python module, that is being imported by third-party projects, and also contains an application—an executable that doesn't interact with Python code (it doesn't even have to be programmed in Python).

If this doesn't fit your package, look into *other sections*.

Table of Contents

Porting the specfile to Python 3

Because the software you're packaging is going to be imported by third-party projects, it is crucial to think about what Python versions your package will support.

If you switch your package to use only Python 3, suddenly projects running on Python 2 will no longer be able to import your modules. And of course, if you continue using Python 2 only, new Python 3 projects won't get to use your software either.

For these reasons, the Fedora Packaging Guidelines for Python advise to **split your package into two subpackages**, one for each major Python version.

In contrast to the Python module, however, the bundled application does not interact with Python code, and is therefore Python version agnostic. For that reason, we need only to include it in one of the subpackages, not both. And when the version does not matter, the guidelines compel us to install the version for Python 3, therefore we will include it in the Python 3 subpackage.

Let's take an example spec file and port it to illustrate the process. We start with a spec file for a Python tool packaged for Python version 2:

```
%global srcname example

Name:           python-%{srcname}
Version:        1.2.3
Release:        1%{?dist}
Summary:        An example Python tool

License:        MIT
URL:            http://pypi.python.org/pypi/%{srcname}
Source0:        https://files.pythonhosted.org/packages/source/e/%{srcname}/%{srcname}
↪-%{version}.tar.gz

BuildArch:      noarch
BuildRequires:  python-devel
Requires: python-some-module
Requires: python2-other-module

%description
A Python tool which provides a convenient example.


%prep
%autosetup -n %{srcname}-%{version}


%build
%{__python} setup.py build


%install
%{__python} setup.py install --skip-build --root $RPM_BUILD_ROOT


%check
%{__python} setup.py test


%files
%license COPYING
%doc README
%{python_sitelib}/*
%{_bindir}/sample-exec


%changelog
...
```

Modifications

First it is recommended to update the software you are packaging to its newest upstream version. If it already is at the latest version, increment the release number. Don't forget to add a `%changelog` entry as well.

Note: In this document you will encounter lot of RPM macros. You can look up many of the Python macros in the

Python packaging guidelines (click the **Expand button** on the right side).

Creating subpackages

Each subpackage you create will need to have its own name, summary and description. If you haven't already, it is thus advised to declare macros for common values at the top of the specfile:

```
%global srcname example
```

Now we can use these to create the subpackages. The following should be placed beneath the `%description` section of the base package:

```
%package -n python2-%{srcname}
Summary:  %{summary}
Requires: python-some-module
Requires: python2-other-module
%{?python_provide:%python_provide python2-%{srcname}}

%description -n python2-%{srcname}
A Python tool which provides a convenient example.


%package -n python3-%{srcname}
Summary:  %{summary}
Requires: python3-some-module
Requires: python3-other-module
%{?python_provide:%python_provide python3-%{srcname}}

%description -n python3-%{srcname}
A Python tool which provides a convenient example.
```

First, using the `%package` macro you start defining a new *subpackage*, specifying its full name as `python2-%{srcname}`, which in this case will become `python2-example`. Next we provide the summary which we defined earlier. `BuildRequires:` tags from the original spec file will remain where they are—declared in the definition of the base package at the top of the spec file. However, the runtime requirements—the ones listed using the `Requires:` tag—will be different for the two subpackages, so they have to be moved here to the definition of each subpackage.

While you can *cut and paste* all the `Requires:` tags directly from the base package to the `python2-` subpackage, remember that for the `python3-` subpackage you need to find Python 3 versions of each of the runtime dependencies.

Note: You can see that the naming of Python 2 packages isn't uniform: some follow the current convention of using the `python2-` prefix, older ones use only the `python-` prefix, and the oldest might be without a prefix at all.

In many cases the Python 2 package can be found under both the `python2-` and `python-` prefixes, one of them being *virtually provided* by the `Provides:` tag. Whenever possible, use the version with the `python2-` prefix.

%python_provide

Now that we're splitting the package `python-example` into `python2-example` and `python3-example`, we need to define what will happen when the user tries to install the unversioned name `python-example`.

At the time of this writing, the packaging guidelines say that the *default* version should be the one for Python 2. However, it is expected to change to Python 3 some time in the future. To avoid having to adjust all Python packages in Fedora when that time comes, the `%python_provide` macro was devised:

```
%{?python_provide:%python_provide python2-%{srcname}}
and
%{?python_provide:%python_provide python3-%{srcname}}
```

This is a line you should include in each of your subpackages and it works thus: First the part `?python_provide:` checks whether the macro exists and if not, the entire line is ignored. After that we actually use the `%python_provide` macro and give it one argument—the name of the given subpackage.

The macro will then check whether this Python version is default or not—if not, the line is again ignored. However, if indeed this is the currently default Python version, the macro is replaced with a *virtual provides* tag: `Provides: python-%{srcname}`. This will tell the packaging system (dnf, yum, ...) to install this subpackage when user searches for `python-example`.

%description

Each subpackage also needs to contain its own description. However, unlike the `Summary:` and `Requires:` tags, which are automatically applied to the subpackage declared above them, the `%description` macro needs to be told to which subpackage it belongs. You can do that by appending the same name as you did with the `%package` macro itself.

```
%description -n python3-%{srcname}
A Python tool which provides a convenient example.
```

BuildRequires and Requires

Now that you're building subpackages for both Python 2 and Python 3, you need to adjust the `BuildRequires:` by adding Python 3 versions of all the existing build dependencies. Starting with `python-devel`: Use its new version-specific name `python2-devel` and add it's Python 3 equivalent `python3-devel`.

As described *above*, `Requires:` tags are a bit more complicated. You should move the current set of `Requires:` underneath the definition of the Python 2 subpackage, and for the Python 3 subpackage, you need to find Python 3 alternatives for all the current Python 2 runtime requirements that are specified with the `Requires:` tags.

As we will be including the executable (application) only in the Python 3 subpackage, you may be also able to get rid of some runtime dependencies (listed using the `Requires:` tags) in the Python 2 subpackage that were previously used only by the executable and are therefore no longer needed in that subpackage. However, figuring out what runtime dependencies are no longer needed is a problematic task, therefore if you are unsure of which dependencies can be omitted, you can skip this task.

%build

Currently your package is building the software for Python 2, what we need to do is also add building for Python 3. While we're modifying the spec file, however, it's a good idea to also update it to new standards—in this case a new macro.

In the ideal case, you'll find the build done with either the `%py2_build` macro or its older version `%py_build`, which you then should exchange for the former. In either case, you can just add the macro `%py3_build` afterwards, and this part is done. Note that to use these macros, you need to have `python2-devel` and/or `python3-devel` listed among `BuildRequires`, but most Python packages already do.

```
%build
%py2_build
%py3_build
```

In many cases, however, you will find a custom build command prefixed by the `%{__python}` or `%{__python2}` macros, or in some cases just prefixed by the python interpreter invoked without a macro at all, e.g.:

```
%{__python} custombuild.py --many-flags
    or
python custombuild.py --many-flags
```

In these cases first try substituting the whole build command by the new pair of smart macros `%py2_build` and `%py3_build`, which should in many cases correctly figure out what ought to be done automatically. Otherwise, duplicate the entire command and change the invocation of the python interpreter to the `%{__python2}` macro in one of them and to the `%{__python3}` in the other.

```
%build
%{__python2} custombuild.py --many-flags
%{__python3} custombuild.py --many-flags
```

Rarely, you might encounter some non-Python build script such as a Makefile. In these instances you have to adjust the script on your own, consult the documentation for the specific build method.

%install

First, in the same manner as in the preceding *%build* section, it is advisable to upgrade the current Python 2 install command to use the new `%py2_install` macro, however, if that doesn't work for you, you can stick with the current install command, just make sure it's invoked by the `%{__python2}` macro.

After the Python 2 install macro is run, it is likely going to install the Python 2 version of the application. As we want to package only the Python 3 version of the application, we have to remove the Python 2 executable(s) that were installed into `/usr/bin/` so that the Python 3 version(s) can take their place afterwards.

```
%install
%py2_install
rm %{buildroot}%{_bindir}/*
```

Note: It is not enough just to run the Python install macros in the right order (first 2 then 3), because the Python installation mechanism (distutils) can sometimes refuse to override files in `/usr/bin`. Even if it works on your machine™, be aware that this might happen on other build systems like Koji or Copr. The issue is also very hard to preduce. That is why it is highly recommended to delete the executables between installs as shown here.

After that, add the corresponding Python 3 install command, which will be either the custom command prefixed by `%{__python3}` or the new `%py3_install` macro.

```
%py3_install
```

Again as in the *%build* section, in the rare cases where you encounter a non-Python install script such as a Makefile, consult documentation for the specific install method and make adjustments on your own.

%check

Unlike in previous sections, there's no special macro for the `%check` section, and so here if the original spec file uses any sort of a python script for testing, just make sure that the tests are invoked once using the `%{__python2}` macro and a second time using the `%{__python3}` macro.

```
%check
%{__python2} setup.py test
%{__python3} setup.py test
```

Chances are that you will encounter a custom Python command that runs the tests, such as `nosetests` or `py.test`. In that case find out what is the name of the executable for Python 3 and run it after the Python 2 command.

If the command for Python 2 can be invoked explicitly for Python 2, e.g. as `py.test-2` instead of just `py.test`, use it. Note that to use py.test commands, you need to have `python2-pytest` and/or `python3-pytest` listed among BuildRequires.

```
%check
py.test-2
py.test-3

or

nosetests-%{python2_version}
nosetests-%{python3_version}
```

As you can see on the example of the `nosetests`, not all packages follow the proper naming conventions for executables. To list what executables a package contains, you can use:

```
$ dnf repoquery -l python3-nose | grep /usr/bin/
/usr/bin/nosetests-3.4
```

%files

The presence or absence of a `%files` section is the deciding factor in whether a given package or subpackage gets built or not. Therefore, to assure that our base package doesn't get built (as all the content has been moved to the two subpackages), make sure there is no `%files` section without a subpackage name.

You can reuse the current `%files` section for the Python 2 submodule by giving it the appropriate package name. You can keep it almost the same as before, just make sure that, where appropriate, it uses the new macros `%{python2_sitelib}`, `%{python2_sitearch}`, `%{python2_version}` or perhaps `%{python2_version_nodots}`.

However, be sure *not to* include the executable. The Fedora Packaging Guidelines for Python state that if you are packaging only one executable, it should be the one for Python 3.

```
%files -n python2-%{srcname}
%license COPYING
%doc README
%{python2_sitelib}/*
```

We'll also add a `%files` section for the Python 3 subpackage. You can copy the previous files section, but make sure you change all the Python 2 macros into Python 3 versions. And in this case, do not forget to *include* the executable as well.

```
%files -n python3-%{srcname}
%license COPYING
%doc README
%{python3_sitelib}/*
%{_bindir}/sample-exec
```

Are shebangs dragging you down (to Python 2)?

A shebang is an indicator on the first line of an executable script that indicates in what interpreter is the script supposed to be launched, examples include *python, bash and perl*. When software gets ported to Python 3, the lone shebang often remains forgotten and keeps pointing to Python 2. In most cases this is handled automatically: the `setup.py` script (usually run through the `%py3_build` and `%py3_install` RPM macros) adjusts shebangs for you. However, sometimes it's up to you to handle the situation.

RPM has very good capabilities of automatically finding dependencies, and one of the ways it accomplishes that is by looking at the shebangs of all the files in the package. Therefore **it is important to check if the shebangs are not dragging in a runtime dependency on Python 2**.

As the porting of the spec file is nearly finished, build it and then run the following analysis on the resulting Python 3 RPM file:

```
$ rpm -qp --requires path/to/an.rpm | grep -E '/usr/bin/(python|env)'
```

This will list all the Python executables your RPM package depends on as well as the `/usr/bin/env` executable which usually invokes `python`. The use of `env` is dangerous: applications should be using the safe system version of Python and not trust whatever version `env` might try to substitute. **If you find that an RPM package for Python 3 depends on Python 2 or `/usr/bin/env` you need to fix it**.

Fixing shebangs

First find out what shebangs are used in your package by unpacking the sources for the project, `cd`-ing into the unpacked directory and trying the following command(s):

```
$ # Searches for all shebangs among the sources
$ grep -r '^#!/' .

$ # Searches only Python shebangs
$ grep -rE '^#!/usr/bin/(python|env python)' .
```

You will usually find one of these two shebangs:

```
#!/usr/bin/python
#!/usr/bin/env python
```

It is advisable to change both of these to `#!/usr/bin/python3`. `/usr/bin/env` can be useful for scripts, but applications should link to the system version of Python outright.

To change the shebangs in the files you can use one (or a combination) of the following commands, which you should place at the end of the `%prep` section. They will change the shebangs to point to the Python 3 interpreter stored in the `${__python3}` macro.

```
$ # Change shebang in individual files
$ sed -i '1s=^#!/usr/bin/\(python\|env python\)[0-9.]*=#!%{__python3}=' path/to/file1␣
↪file2 file3 ...
```

```
$ # Change shebang in all relevant files in this directory and all subdirectories
$ # See `man find` for how the `-exec command {} +` syntax works
$ find -type f -exec sed -i '1s=^#!/usr/bin/\(python\|env python\)[23]\?=#!%{__
↪python3}=' {} +

$ # Change shebang in all relevant executable files in this directory and all␣
↪subdirectories
$ find -type f -executable -exec sed -i '1s=^#!/usr/bin/\(python\|env python\)[23]\?=
↪#!%{__python3}=' {} +
```

You don't have to worry about accidentally corrupting other files as these scripts will only change a file if the beginning of its first line exactly matches one of the two aforementioned shebangs.

Have you broken third-party packages?

Congratulations, you have now successfully ported your package to be available for both Python 2 and Python 3! However, in doing so, many of the third-party packages that depend on the application bundled in this package may have just been broken.

The best practice when depending on executables is to depend on them explicitly, i.e. use `Requires: /usr/bin/sample-exec`. That way no matter to which package the executable moves, your dependency gets loaded fine. However, many (if not most) packages are written with dependencies on the package itself (`Requires: python-example`) in which case they will now be depending on the `python2-example` subpackage, because it has the currently active `%python_provide` macro (see *%python_provide*). However, the executable has moved to the `python3-example` subpackage, and thus the dependency has been broken.

First, see what (if any) packages depend on this package itself:

```
$ dnf repoquery --whatrequires python-example
```

Now you ought to go through these packages one by one and try to figure out if they need to depend on the application from your package, or on the Python module, or possibly, both.

If you do think they want to depend on your application, and therefore the dependency may have just been broken, you are advised to open a BugZilla report and request that they change (or add) the dependency to the executable itself (`Requires: /usr/bin/sample-exec`). If you can provide a patch as well, your requests will be all the faster resolved.

If you are unsure whether the package needs to depend on your application, open a BugZilla report for the package and ask the maintainer(s) to answer the question themselves.

Ported RPM spec file

Here you can peruse the entire ported spec file:

```
%global srcname example

Name:           python-%{srcname}
Version:        1.2.3
Release:        2%{?dist}
Summary:        An example Python tool

License:        MIT
URL:            http://pypi.python.org/pypi/%{srcname}
Source0:        https://files.pythonhosted.org/packages/source/e/%{srcname}/%{srcname}
↪-%{version}.tar.gz
```

```
BuildArch:      noarch
BuildRequires:  python2-devel
BuildRequires:  python3-devel

%description
A Python tool which provides a convenient example.


%package -n python2-%{srcname}
Summary:        %{summary}
Requires:       python-some-module
Requires:       python2-other-module
%{?python_provide:%python_provide python2-%{srcname}}

%description -n python2-%{srcname}
A Python tool which provides a convenient example.


%package -n python3-%{srcname}
Summary:        %{summary}
Requires:       python3-some-module
Requires:       python3-other-module
%{?python_provide:%python_provide python3-%{srcname}}

%description -n python3-%{srcname}
A Python tool which provides a convenient example.


%prep
%autosetup -n %{srcname}-%{version}


%build
%py2_build
%py3_build


%install
%py2_install

# The Python 2 installation process will likely try to install its own version
# of the application. As we only want to package the Python 3 version of the
# application, we delete the Python 2 executable(s) so that the Python 3
# version(s) can take their place afterwards.
rm %{buildroot}%{_bindir}/*

%py3_install


%check
%{__python2} setup.py test
%{__python3} setup.py test


# Note that there is no %%files section for the unversioned Python package
# if we are building for several Python runtimes
%files -n python2-%{srcname}
```

```
%license COPYING
%doc README
%{python2_sitelib}/*

%files -n python3-%{srcname}
%license COPYING
%doc README
%{python3_sitelib}/*
%{_bindir}/sample-exec


%changelog
...
```

Diff of the changes

And here you can see the diff of the original and the ported spec files to fully observe all the changes that were made:

```
--- specs/tool.spec.orig
+++ specs/application-module.spec
@@ -2,7 +2,7 @@

 Name:           python-%{srcname}
 Version:        1.2.3
-Release:        1%{?dist}
+Release:        2%{?dist}
 Summary:        An example Python tool

 License:        MIT
@@ -10,11 +10,30 @@
 Source0:        https://files.pythonhosted.org/packages/source/e/%{srcname}/%
↪{srcname}-%{version}.tar.gz

 BuildArch:      noarch
-BuildRequires:  python-devel
-Requires: python-some-module
-Requires: python2-other-module
+BuildRequires:  python2-devel
+BuildRequires:  python3-devel

 %description
+A Python tool which provides a convenient example.
+
+
+%package -n python2-%{srcname}
+Summary:        %{summary}
+Requires:       python-some-module
+Requires:       python2-other-module
+%{?python_provide:%python_provide python2-%{srcname}}
+
+%description -n python2-%{srcname}
+A Python tool which provides a convenient example.
+
+
+%package -n python3-%{srcname}
+Summary:        %{summary}
```

```
+Requires:       python3-some-module
+Requires:       python3-other-module
+%{?python_provide:%python_provide python3-%{srcname}}
+
+%description -n python3-%{srcname}
 A Python tool which provides a convenient example.


@@ -23,21 +42,38 @@


 %build
-%{__python} setup.py build
+%py2_build
+%py3_build


 %install
-%{__python} setup.py install --skip-build --root $RPM_BUILD_ROOT
+%py2_install
+
+# The Python 2 installation process will likely try to install its own version
+# of the application. As we only want to package the Python 3 version of the
+# application, we delete the Python 2 executable(s) so that the Python 3
+# version(s) can take their place afterwards.
+rm %{buildroot}%{_bindir}/*
+
+%py3_install


 %check
-%{__python} setup.py test
+%{__python2} setup.py test
+%{__python3} setup.py test


-%files
+# Note that there is no %%files section for the unversioned Python package
+# if we are building for several Python runtimes
+%files -n python2-%{srcname}
 %license COPYING
 %doc README
-%{python_sitelib}/*
+%{python2_sitelib}/*
+
+%files -n python3-%{srcname}
+%license COPYING
+%doc README
+%{python3_sitelib}/*
 %{_bindir}/sample-exec
```

Tools for programming in Python

This last section is for tools (executables) written in Python that themselves interact with Python code. If you need to package two versions of the executable, one for each major Python version, then this section is for you.

Examples: *pip, pytest, nosetest*.

If this doesn't fit your package, look into *other sections*.

Table of Contents

Porting the specfile to Python 3

Because your package has an executable that interacts with Python code, users will most likely want it to work both with code for Python 2 and code for Python 3, as both are very common today. And even in the future, there will be heaps of legacy Python 2 code that may still be useful.

When you want to package two different executables, the best practice (put forth by the Fedora Packaging Guidelines for Python) is to **split your package into two subpackages.** One subpackage for Python 2 and one for Python 3. That will allow the user to install just one of them instead of having to install the other as well, for which they not have any need.

Let's take an example spec file and port it to illustrate the process. We start with a spec file for a Python tool packaged for Python version 2:

```
%global srcname example

Name:           python-%{srcname}
Version:        1.2.3
Release:        1%{?dist}
```

```
Summary:        An example Python tool

License:        MIT
URL:            http://pypi.python.org/pypi/%{srcname}
Source0:        https://files.pythonhosted.org/packages/source/e/%{srcname}/%{srcname}
↪-%{version}.tar.gz

BuildArch:      noarch
BuildRequires:  python-devel
Requires: python-some-module
Requires: python2-other-module

%description
A Python tool which provides a convenient example.


%prep
%autosetup -n %{srcname}-%{version}


%build
%{__python} setup.py build


%install
%{__python} setup.py install --skip-build --root $RPM_BUILD_ROOT


%check
%{__python} setup.py test


%files
%license COPYING
%doc README
%{python_sitelib}/*
%{_bindir}/sample-exec


%changelog
...
```

Modifications

First it is recommended to update the software you are packaging to its newest upstream version. If it already is at the latest version, increment the release number. Don't forget to add a `%changelog` entry as well.

Note: In this document you will encounter lot of RPM macros. You can look up many of the Python macros in the Python packaging guidelines (click the **Expand button** on the right side).

Creating subpackages

Each subpackage you create will need to have its own name, summary and description. If you haven't already, it is thus advised to declare macros for common values at the top of the specfile:

```
%global srcname example
```

Now we can use these to create the subpackages. The following should be placed beneath the `%description` section of the base package:

```
%package -n python2-%{srcname}
Summary:  %{summary}
Requires: python-some-module
Requires: python2-other-module
%{?python_provide:%python_provide python2-%{srcname}}

%description -n python2-%{srcname}
A Python tool which provides a convenient example.


%package -n python3-%{srcname}
Summary:  %{summary}
Requires: python3-some-module
Requires: python3-other-module
%{?python_provide:%python_provide python3-%{srcname}}

%description -n python3-%{srcname}
A Python tool which provides a convenient example.
```

First, using the `%package` macro you start defining a new *subpackage*, specifying its full name as `python2-%{srcname}`, which in this case will become `python2-example`. Next we provide the summary which we defined earlier. `BuildRequires:` tags from the original spec file will remain where they are—declared in the definition of the base package at the top of the spec file. However, the runtime requirements—the ones listed using the `Requires:` tag—will be different for the two subpackages, so they have to be moved here to the definition of each subpackage.

While you can *cut and paste* all the `Requires:` tags directly from the base package to the `python2-` subpackage, remember that for the `python3-` subpackage you need to find Python 3 versions of each of the runtime dependencies.

Note: You can see that the naming of Python 2 packages isn't uniform: some follow the current convention of using the `python2-` prefix, older ones use only the `python-` prefix, and the oldest might be without a prefix at all.

In many cases the Python 2 package can be found under both the `python2-` and `python-` prefixes, one of them being *virtually provided* by the `Provides:` tag. Whenever possible, use the version with the `python2-` prefix.

%python_provide

Now that we're splitting the package `python-example` into `python2-example` and `python3-example`, we need to define what will happen when the user tries to install the unversioned name `python-example`.

At the time of this writing, the packaging guidelines say that the *default* version should be the one for Python 2. However, it is expected to change to Python 3 some time in the future. To avoid having to adjust all Python packages in Fedora when that time comes, the `%python_provide` macro was devised:

```
%{?python_provide:%python_provide python2-%{srcname}}
and
%{?python_provide:%python_provide python3-%{srcname}}
```

This is a line you should include in each of your subpackages and it works thus: First the part `?python_provide:` checks whether the macro exists and if not, the entire line is ignored. After that we actually use the `%python_provide` macro and give it one argument—the name of the given subpackage.

The macro will then check whether this Python version is default or not—if not, the line is again ignored. However, if indeed this is the currently default Python version, the macro is replaced with a *virtual provides* tag: `Provides: python-%{srcname}`. This will tell the packaging system (dnf, yum, ...) to install this subpackage when user searches for `python-example`.

%description

Each subpackage also needs to contain its own description. However, unlike the `Summary:` and `Requires:` tags, which are automatically applied to the subpackage declared above them, the `%description` macro needs to be told to which subpackage it belongs. You can do that by appending the same name as you did with the `%package` macro itself.

```
%description -n python3-%{srcname}
A Python tool which provides a convenient example.
```

BuildRequires and Requires

Now that you're building subpackages for both Python 2 and Python 3, you need to adjust the `BuildRequires:` by adding Python 3 versions of all the existing build dependencies. Starting with `python-devel`: Use its new version-specific name `python2-devel` and add it's Python 3 equivalent `python3-devel`.

As described *above*, `Requires:` tags are a bit more complicated. You should move the current set of `Requires:` underneath the definition of the Python 2 subpackage, and for the Python 3 subpackage, you need to find Python 3 alternatives for all the current Python 2 runtime requirements that are specified with the `Requires:` tags.

%build

Currently your package is building the software for Python 2, what we need to do is also add building for Python 3. While we're modifying the spec file, however, it's a good idea to also update it to new standards—in this case a new macro.

In the ideal case, you'll find the build done with either the `%py2_build` macro or its older version `%py_build`, which you then should exchange for the former. In either case, you can just add the macro `%py3_build` afterwards, and this part is done. Note that to use these macros, you need to have `python2-devel` and/or `python3-devel` listed among `BuildRequires`, but most Python packages already do.

```
%build
%py2_build
%py3_build
```

In many cases, however, you will find a custom build command prefixed by the `%{__python}` or `%{__python2}` macros, or in some cases just prefixed by the python interpreter invoked without a macro at all, e.g.:

```
%{__python} custombuild.py --many-flags
    or
python custombuild.py --many-flags
```

In these cases first try substituting the whole build command by the new pair of smart macros `%py2_build` and `%py3_build`, which should in many cases correctly figure out what ought to be done automatically. Otherwise, duplicate the entire command and change the invocation of the python interpreter to the `%{__python2}` macro in one of them and to the `%{__python3}` in the other.

```
%build
%{__python2} custombuild.py --many-flags
%{__python3} custombuild.py --many-flags
```

Rarely, you might encounter some non-Python build script such as a Makefile. In these instances you have to adjust the script on your own, consult the documentation for the specific build method.

%install

The `%install` section is perhaps the most crucial one, because we have to be very mindful of which executable goes where and what symlinks should be created.

First, in the same manner as in the preceding *%build* section, it is advisable to upgrade the current Python 2 install command to use the new `%py2_install` macro, however, if that doesn't work for you, you can stick with the current install command, just make sure it's invoked by the `%{__python2}` macro. The corresponding Python 3 install command will then either be the custom command prefixed by `%{__python3}` or the new `%py3_install` macro, which I'll be using in this example.

Again as in the *%build* section, in the rare cases where you encounter a non-Python install script such as a Makefile, consult documentation for the specific install method and make adjustments on your own.

As the packaging guidelines specify, the Python 2 package is currently to be the default one, thus it is best if we first install the Python 3 version of our software and then the one for Python 2, because in case they are installing some files into the same directories (such as `/usr/bin/`), one installation will overwrite the files of the other. So if we install the Python 2 version last, its files will be located in those shared directories.

```
%install
%py3_install

# Now /usr/bin/sample-exec is Python 3, so we move it away
mv %{buildroot}%{_bindir}/sample-exec %{buildroot}%{_bindir}/sample-exec-%{python3_
↪version}

%py2_install

# Now /usr/bin/sample-exec is Python 2, and we move it away anyway
mv %{buildroot}%{_bindir}/sample-exec %{buildroot}%{_bindir}/sample-exec-%{python2_
↪version}
```

Moving the Executables and Making Symlinks

Again in compliance with the packaging guidelines, we should provide the executables in the format `executable-name-X.Y`, where X and Y are the major and minor Python versions, e.g. `2.7` or `3.5` (instead of hardcoding these values, you can use the macros `%{python2_version}` and `%{python3_version}`). Thus after each of the install scripts finishes, we move the resulting executable and rename it accordingly.

What remains is to provide symlinks for the executables in the format `executable-name-2` and `executable-name-3`, pointing to their respective executables, and then finally a symlink for the general name, in this case `sample-exec`, which shall point to the Python 2 version symlink—here `sample-exec-2`.

```
# The guidelines also specify we must provide symlinks with a '-X' suffix.
ln -s ./sample-exec-%{python2_version} %{buildroot}%{_bindir}/sample-exec-2
ln -s ./sample-exec-%{python3_version} %{buildroot}%{_bindir}/sample-exec-3

# Finally, we provide /usr/bin/sample-exec as a link to /usr/bin/sample-exec-2
ln -s ./sample-exec-2 %{buildroot}%{_bindir}/sample-exec
```

Note that these symlinks use a relative path in relation to their location, i.e. they are pointing to a file that is at any given moment in the same directory as they are.

%check

Unlike in previous sections, there's no special macro for the `%check` section, and so here if the original spec file uses any sort of a python script for testing, just make sure that the tests are invoked once using the `%{__python2}` macro and a second time using the `%{__python3}` macro.

```
%check
%{__python2} setup.py test
%{__python3} setup.py test
```

Chances are that you will encounter a custom Python command that runs the tests, such as `nosetests` or `py.test`. In that case find out what is the name of the executable for Python 3 and run it after the Python 2 command.

If the command for Python 2 can be invoked explicitly for Python 2, e.g. as `py.test-2` instead of just `py.test`, use it. Note that to use py.test commands, you need to have `python2-pytest` and/or `python3-pytest` listed among BuildRequires.

```
%check
py.test-2
py.test-3

or

nosetests-%{python2_version}
nosetests-%{python3_version}
```

As you can see on the example of the `nosetests`, not all packages follow the proper naming conventions for executables. To list what executables a package contains, you can use:

```
$ dnf repoquery -l python3-nose | grep /usr/bin/
/usr/bin/nosetests-3.4
```

%files

The presence or absence of a `%files` section is the deciding factor in whether a given package or subpackage gets built or not. Therefore, to assure that our base package doesn't get built (as all the content has been moved to the two subpackages), make sure there is no `%files` section without a subpackage name.

You can reuse the current `%files` section for the Python 2 submodule by giving it the appropriate package name. You can keep it almost the same as before, just make sure that, where appropriate, it uses the new macros `%{python2_sitelib}`, `%{python2_sitearch}`, `%{python2_version}` or perhaps

`%{python2_version_nodots}`. Finally, don't forget to add the two new locations of the executable we've made available through the symlinks.

```
%files -n python2-%{srcname}
%license COPYING
%doc README
%{python2_sitelib}/*
%{_bindir}/sample-exec
%{_bindir}/sample-exec-2
%{_bindir}/sample-exec-%{python2_version}
```

Accordingly we'll also add a `%files` section for the Python 3 subpackage. You can copy the previous files section, but make sure you change all the Python 2 macros into Python 3 versions. And unlike the former, the Python 3 `%files` section shall not contain the unversioned executable (`sample-exec` in our example) as that executable is for Python 2, not 3.

```
%files -n python3-%{srcname}
%license COPYING
%doc README
%{python3_sitelib}/*
%{_bindir}/sample-exec-3
%{_bindir}/sample-exec-%{python3_version}
```

Are shebangs dragging you down (to Python 2)?

A shebang is an indicator on the first line of an executable script that indicates in what interpreter is the script supposed to be launched, examples include *python, bash and perl*. When software gets ported to Python 3, the lone shebang often remains forgotten and keeps pointing to Python 2. In most cases this is handled automatically: the `setup.py` script (usually run through the `%py3_build` and `%py3_install` RPM macros) adjusts shebangs for you. However, sometimes it's up to you to handle the situation.

RPM has very good capabilities of automatically finding dependencies, and one of the ways it accomplishes that is by looking at the shebangs of all the files in the package. Therefore **it is important to check if the shebangs are not dragging in a runtime dependency on Python 2**.

As the porting of the spec file is nearly finished, build it and then run the following analysis on the resulting Python 3 RPM file:

```
$ rpm -qp --requires path/to/an.rpm | grep -E '/usr/bin/(python|env)'
```

This will list all the Python executables your RPM package depends on as well as the `/usr/bin/env` executable which usually invokes `python`. The use of `env` is dangerous: applications should be using the safe system version of Python and not trust whatever version `env` might try to substitute. **If you find that an RPM package for Python 3 depends on Python 2 or `/usr/bin/env` you need to fix it**.

Fixing shebangs

First find out what shebangs are used in your package by unpacking the sources for the project, `cd`-ing into the unpacked directory and trying the following command(s):

```
$ # Searches for all shebangs among the sources
$ grep -r '^#!/' .

$ # Searches only Python shebangs
$ grep -rE '^#!/usr/bin/(python|env python)' .
```

You will usually find one of these two shebangs:

```
#!/usr/bin/python
#!/usr/bin/env python
```

It is advisable to change both of these to `#!/usr/bin/python3`. `/usr/bin/env` can be useful for scripts, but applications should link to the system version of Python outright.

To change the shebangs in the files you can use one (or a combination) of the following commands, which you should place at the end of the `%prep` section. They will change the shebangs to point to the Python 3 interpreter stored in the `${__python3}` macro.

```
$ # Change shebang in individual files
$ sed -i '1s=^#!/usr/bin/\(python\|env python\)[0-9.]*=#!%{__python3}=' path/to/file1␣
↪file2 file3 ...

$ # Change shebang in all relevant files in this directory and all subdirectories
$ # See `man find` for how the `-exec command {} +` syntax works
$ find -type f -exec sed -i '1s=^#!/usr/bin/\(python\|env python\)[23]\?=#!%{__
↪python3}=' {} +

$ # Change shebang in all relevant executable files in this directory and all␣
↪subdirectories
$ find -type f -executable -exec sed -i '1s=^#!/usr/bin/\(python\|env python\)[23]\?=
↪#!%{__python3}=' {} +
```

You don't have to worry about accidentally corrupting other files as these scripts will only change a file if the beginning of its first line exactly matches one of the two aforementioned shebangs.

Ported RPM spec file

Here you can peruse the entire ported spec file:

```
%global srcname example

Name:           python-%{srcname}
Version:        1.2.3
Release:        2%{?dist}
Summary:        An example Python tool

License:        MIT
URL:            http://pypi.python.org/pypi/%{srcname}
Source0:        https://files.pythonhosted.org/packages/source/e/%{srcname}/%{srcname}
↪-%{version}.tar.gz

BuildArch:      noarch
BuildRequires:  python2-devel
BuildRequires:  python3-devel

%description
A Python tool which provides a convenient example.


%package -n python2-%{srcname}
Summary:        %{summary}
Requires:       python-some-module
Requires:       python2-other-module
```

```
%{?python_provide:%python_provide python2-%{srcname}}

%description -n python2-%{srcname}
A Python tool which provides a convenient example.


%package -n python3-%{srcname}
Summary:        %{summary}
Requires:       python3-some-module
Requires:       python3-other-module
%{?python_provide:%python_provide python3-%{srcname}}

%description -n python3-%{srcname}
A Python tool which provides a convenient example.


%prep
%autosetup -n %{srcname}-%{version}


%build
%py2_build
%py3_build


%install
# Here we have to think about the order, because the scripts in /usr/bin are
# overwritten with every setup.py install.
# If the script in /usr/bin provides the same functionality regardless
# of the Python version, we only provide Python 3 version and we need to run
# the py3_install after py2_install.

# If we need to include the executable both for Python 2 and 3--for example
# because it interacts with code from the user--then the default executable
# should be the one for Python 2.
# We are going to assume that case here, because it is a bit more complex.

%py3_install

# Now /usr/bin/sample-exec is Python 3, so we move it away
mv %{buildroot}%{_bindir}/sample-exec %{buildroot}%{_bindir}/sample-exec-%{python3_
↪version}

%py2_install

# Now /usr/bin/sample-exec is Python 2, and we move it away anyway
mv %{buildroot}%{_bindir}/sample-exec %{buildroot}%{_bindir}/sample-exec-%{python2_
↪version}

# The guidelines also specify we must provide symlinks with a '-X' suffix.
ln -s ./sample-exec-%{python2_version} %{buildroot}%{_bindir}/sample-exec-2
ln -s ./sample-exec-%{python3_version} %{buildroot}%{_bindir}/sample-exec-3

# Finally, we provide /usr/bin/sample-exec as a link to /usr/bin/sample-exec-2
ln -s ./sample-exec-2 %{buildroot}%{_bindir}/sample-exec


%check
```

```
%{__python2} setup.py test
%{__python3} setup.py test


# Note that there is no %%files section for the unversioned Python package
# if we are building for several Python runtimes
%files -n python2-%{srcname}
%license COPYING
%doc README
%{python2_sitelib}/*
%{_bindir}/sample-exec
%{_bindir}/sample-exec-2
%{_bindir}/sample-exec-%{python2_version}

%files -n python3-%{srcname}
%license COPYING
%doc README
%{python3_sitelib}/*
%{_bindir}/sample-exec-3
%{_bindir}/sample-exec-%{python3_version}


%changelog
...
```

Diff of the changes

And here you can see the diff of the original and the ported spec files to fully observe all the changes that were made:

```
--- specs/tool.spec.orig
+++ specs/tool.spec
@@ -2,7 +2,7 @@

 Name:           python-%{srcname}
 Version:        1.2.3
-Release:        1%{?dist}
+Release:        2%{?dist}
 Summary:        An example Python tool

 License:        MIT
@@ -10,11 +10,30 @@
 Source0:        https://files.pythonhosted.org/packages/source/e/%{srcname}/%
↪{srcname}-%{version}.tar.gz

 BuildArch:      noarch
-BuildRequires:  python-devel
-Requires: python-some-module
-Requires: python2-other-module
+BuildRequires:  python2-devel
+BuildRequires:  python3-devel

 %description
+A Python tool which provides a convenient example.
+
+
+%package -n python2-%{srcname}
```

```
+Summary:        %{summary}
+Requires:       python-some-module
+Requires:       python2-other-module
+%{?python_provide:%python_provide python2-%{srcname}}
+
+%description -n python2-%{srcname}
+A Python tool which provides a convenient example.
+
+
+%package -n python3-%{srcname}
+Summary:        %{summary}
+Requires:       python3-some-module
+Requires:       python3-other-module
+%{?python_provide:%python_provide python3-%{srcname}}
+
+%description -n python3-%{srcname}
 A Python tool which provides a convenient example.


@@ -23,22 +42,61 @@


 %build
-%{__python} setup.py build
+%py2_build
+%py3_build


 %install
-%{__python} setup.py install --skip-build --root $RPM_BUILD_ROOT
+# Here we have to think about the order, because the scripts in /usr/bin are
+# overwritten with every setup.py install.
+# If the script in /usr/bin provides the same functionality regardless
+# of the Python version, we only provide Python 3 version and we need to run
+# the py3_install after py2_install.
+
+# If we need to include the executable both for Python 2 and 3--for example
+# because it interacts with code from the user--then the default executable
+# should be the one for Python 2.
+# We are going to assume that case here, because it is a bit more complex.
+
+%py3_install
+
+# Now /usr/bin/sample-exec is Python 3, so we move it away
+mv %{buildroot}%{_bindir}/sample-exec %{buildroot}%{_bindir}/sample-exec-%{python3_
↪version}
+
+%py2_install
+
+# Now /usr/bin/sample-exec is Python 2, and we move it away anyway
+mv %{buildroot}%{_bindir}/sample-exec %{buildroot}%{_bindir}/sample-exec-%{python2_
↪version}
+
+# The guidelines also specify we must provide symlinks with a '-X' suffix.
+ln -s ./sample-exec-%{python2_version} %{buildroot}%{_bindir}/sample-exec-2
+ln -s ./sample-exec-%{python3_version} %{buildroot}%{_bindir}/sample-exec-3
+
+# Finally, we provide /usr/bin/sample-exec as a link to /usr/bin/sample-exec-2
```

```
+ln -s ./sample-exec-2 %{buildroot}%{_bindir}/sample-exec


 %check
-%{__python} setup.py test
+%{__python2} setup.py test
+%{__python3} setup.py test


-%files
+# Note that there is no %%files section for the unversioned Python package
+# if we are building for several Python runtimes
+%files -n python2-%{srcname}
 %license COPYING
 %doc README
-%{python_sitelib}/*
+%{python2_sitelib}/*
 %{_bindir}/sample-exec
+%{_bindir}/sample-exec-2
+%{_bindir}/sample-exec-%{python2_version}
+
+%files -n python3-%{srcname}
+%license COPYING
+%doc README
+%{python3_sitelib}/*
+%{_bindir}/sample-exec-3
+%{_bindir}/sample-exec-%{python3_version}


 %changelog
```

New Python package naming scheme

If you have ported your package according to the guidance in previous sections, then it most certainly complies with the Python package naming guidelines. However, you might have done it before this guide came to life or even before the naming guidelines changed, which might mean that some name changes are required. This section is here to walk you through them.

Note: This section is related to the renaming of Python **binary RPM** packages to avoid using the `python-` prefix without a version. Changing the main package/SRPM name is not required.

Why is this important?

When the time comes and `python` means Python 3 in Fedora, installing a `python-<srcname>` package will imply a Python 3 version of this package. This is planned for 2020, when upstream support for Python 2 ends. To achieve this, all Python binary RPM packages have to follow the new naming scheme and use the `%python_provide` macro, devised to make the switch easier. However, we are still far away from achieving this goal.

Using the outdated naming scheme in your subpackage names or run-time/build-time requirements might cause a range of issues when the switch happens.

What needs to be changed?

Check the names of binary RPM packages you are building from your SRPM, and if you use one of the following naming schemes, you'll find instructions on how to fix it in the *section below*.

Common naming scheme violations

See the following naming schemes which violate the current naming guidelines:

SRPM	Binary RPMs built	Violation
python-<srcname>	python-<srcname>	unversioned *python-* prefix in the binary package
python-<srcname>	python-<srcname> python3-<srcname>	unversioned *python-* prefix in the binary package
<srcname>	<srcname> python3-<srcname>	missing *python2-* prefix in the binary package

Required changes

Add a `%package` section for the Python 2 subpackage

To rename the binary RPM package to `python2-<srcname>`, you should build it as a subpackage with an appropriate name. Make sure to move all related runtime requirements from the main package to the new subpackage and use the %python_provide macro, which will provide both `python-<srcname>` and `python2-<srcname>` until the switch to Python 3 happens.

The change should look like this:

```
 BuildArch:      noarch
-BuildRequires:  python-devel
-Requires:       python-some-module
-Requires:       python2-other-module
+BuildRequires:  python2-devel

 %description
+A Python module which provides a convenient example.
+
+
+%package -n python2-%{srcname}
+Summary:        %{summary}
+Requires:       python2-some-module
+Requires:       python2-other-module
+%{?python_provide:%python_provide python2-%{srcname}}
+
+%description -n python2-%{srcname}
 A Python module which provides a convenient example.
```

Note, that in case of the last naming scheme example in the *table above*, when you rename the binary RPM from `<srcname>` to `python2-<srcname>`, the %python_provide macro will not provide the old name `<srcname>`. To keep the upgrade path clean you will have to provide it and obsolete the old verision manually. You may place the tags right after the %python_provide macro:

```
%{?python_provide:%python_provide python2-%{srcname}}
Provides:   %{srcname} = %{version}-%{release}
Obsoletes:  %{srcname} < current_version-current_release
```

In the Obsoletes tag, `current_version` and `current_release` are the first version and release when the new naming scheme was used.

Use the `%python_provide` macro in the Python 3 subpackage

Check your Python 3 subpackage (if you build any), and make sure you are using the %python_provide macro to handle provides:

```
%package -n python3-%{srcname}
Summary:        %{summary}
Requires:       python3-some-module
Requires:       python3-other-module
%{?python_provide:%python_provide python3-%{srcname}}

%description -n python3-%{srcname}
A Python module which provides a convenient example.
```

Rename the `%files` section

To assign the `%files` section to the Python 2 subpackage, add the subpackage name with the versioned prefix after the `%files` macro. Make sure to use the new versioned macros `%{python2_sitelib}`, `%{python2_sitearch}`, and `%{python2_version}` as well:

```
-%files
+# Note that there is no %%files section for the unversioned Python package
+%files -n python2-%{srcname}
 %license COPYING
 %doc README
-%{python_sitelib}/*
+%{python2_sitelib}/*
```

At this point you should be done. Don't forget to bump the release tag and add a changelog entry indicating you've updated the package to use the new naming scheme.

Welcome to the Python RPM Porting Guide

This document aims to guide you through the process of porting your Python 2 RPM package to Python 3.

Note: If you struggle with the porting of your package and would like help or more information, please contact us at the python-devel mailing list (also accessible through a web interface) and we will try to help you and/or improve the guide accordingly.

If you've spotted any errors, have any suggestions or think some section(s) could be expanded, please create an Issue or a Pull request on our GitHub.

Table of Contents

Is your package ready to be ported?

First thing you need to figure out is if the software you're packaging is ready to be packaged for Python 3.

Does upstream support Python 3?

Look upstream and try to find out if the software is released with Python 3 support. First look at the front page of the project, Python compatibility is oftentimes listed there. There is also a good chance the project will have this information listed on PyPI. If not, look at release notes or the changelog history. You can also look at issues and pull requests.

However, the important thing to note is that the Python 3 support needs to be *released*, not just committed in the version control system (git, mercurial,...).

Are the dependencies of your package ported to Python 3 for your distribution?

Before you start porting, it's imperative that you check that all the dependencies of your package are also ported to Python 3 in the distribution you are packaging for (Fedora, CentOS, RHEL or any other RPM-based distribution).

You may encounter a situation where your software is Python 3 ready upstream, but it uses some dependencies that are packaged only for Python 2 in your distribution. In that case try to communicate with the maintainer of the needed package and try to motivate and/or help them with porting of the package.

What type of software are you packaging?

There are four distinct types of Python packages, each with different instructions for porting, so be mindful of which you chose.

In rare cases your package might not nicely fall into either of these categories. In that case use the relevant parts from multiple sections or contact us on the mailing list.

1. Applications that happen to be written in Python

The section for applications is for software where the user doesn't care in which programming language it is written. For example, it doesn't matter if it is written in Python 2 or 3 because the application should run the same.

Applications have to have an executable, which you can check by running `dnf repoquery -l your-package-name | grep /usr/bin/`.

If your package is not being imported by third-party projects (e.g. `import some_module` in a Python file), it is most likely an application.

Try running `dnf repoquery --whatrequires your-package-name` to see a list of packages that depend on yours. If there are none, your package is likely an application, because there would be little reason to package a module that nobody uses. However, if there are some packages that depend on yours, we cannot be sure if it's an application-only package or not, as some of these packages might be depending on your application itself, instead of importing modules from your package.

See: Porting applications that happen to be written in Python

2. Python modules

If your package is being imported by third-party projects, but does not have any executables, you're dealing with a Python module.

See: Porting Python modules

3. An application and a module in one package

Use this section if your package contains both a Python module, that is being imported by third-party projects, and also contains an application—an executable that doesn't interact with Python code (it doesn't even have to be programmed in Python).

See: Porting an application and a module in one package

4. Tools for programming in Python

This last section is for tools (executables) written in Python that themselves interact with Python code. If you need to package two versions of the executable, one for each major Python version, then this section is for you.

Examples: *pip, pytest, nosetest*.

See: Tools for programming in Python

Are you following the new Python package naming scheme?

If you have ported your package according to the guidance in previous sections, then it most certainly complies with the Python package naming guidelines. However, you might have done it before this guide came to life or even before the naming guidelines changed, which might mean that some name changes are required. This section is here to walk you through them.

See: New Python package naming scheme