Red Hat Enterprise Linux 9

動的プログラミング言語のインストールおよび使用

Red Hat Enterprise Linux 9 で動的プログラミング言語をインストールおよび使用するためのガイド

Last Updated: 2022-05-27

Red Hat Enterprise Linux 9 動的プログラミング言語のインストールおよび使用

Red Hat Enterprise Linux 9 で動的プログラミング言語をインストールおよび使用するためのガイド

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

法律上の通知

概要

本書は、Red Hat Enterprise Linux 9 での Python や PHP などの動的プログラミング言語のインストールと使用の基本について説明します。

目次

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、弊社の CTO、Chris Wright のメッセージ を参照してください。

RED HAT ドキュメントへのフィードバックの提供

ご意見ご要望をお聞かせください。ドキュメントの改善点はございませんか。

- 特定の部分についての簡単なコメントをお寄せいただく場合は、以下をご確認ください。
    1. ドキュメントの表示が **Multi-page HTML** 形式になっていて、ドキュメントの右上隅に **Feedback** ボタンがあることを確認してください。
    2. マウスカーソルで、コメントを追加する部分を強調表示します。
    3. そのテキストの下に表示される **Add Feedback** ポップアップをクリックします。
    4. 表示される手順に従ってください。
- Bugzilla を介してフィードバックを送信するには、新しいチケットを作成します。
    1. Bugzilla の Web サイトに移動します。
    2. Component で **Documentation** を選択します。
    3. **Description** フィールドに、ドキュメントの改善に関するご意見を記入してください。ドキュメントの該当部分へのリンクも記入してください。
    4. **Submit Bug** をクリックします。

第1章 PYTHON の概要

Python は、オブジェクト指向、命令、機能、手順などの複数のプログラミングパラダイムをサポートする、高レベルのプログラミング言語です。Python は動的なセマンティクスを持ち、汎用プログラミングに使用できます。

Red Hat Enterprise Linux では、システムツール、データ分析用のツール、Web アプリケーションを提供するパッケージなど、システムにインストールされている多くのパッケージが Python で記述されています。このパッケージを使用するには、**python*** パッケージがインストールされている必要があります。

1.1. PYTHON のバージョン

Python 3.9 が RHEL 9 におけるデフォルトの **Python** 実装に**Python 3.9** は、BaseOS リポジトリーにあるモジュール以外の **python3** RPM パッケージで配布され、通常はデフォルトでインストールされます。**Python 3.9** は、RHEL 9 のライフサイクル全体で対応します。

今後は、AppStream リポジトリーを介して、追加のバージョンの **Python 3** が RPM パッケージとして配布され、ライフサイクルが短くなります。これらのバージョンは、Python 3.9 と並行してインストールできます。

Python 2 は RHEL 9 に同梱されていません。

1.2. RHEL 8 以降の PYTHON エコシステムの主な相違点

本セクションでは、RHEL 8 と比較した RHEL 9 の Python エコシステムの主な変更をまとめています。

バージョンを指定しない python コマンド

バージョンを指定しない**python** コマンド (**/usr/bin/python**) は、**python-unversioned-command** パッケージで利用できます。一部のシステムでは、このパッケージはデフォルトでインストールされていません。バージョンを指定しない **python** コマンドを手動でインストールする場合は、**dnf install /usr/bin/python** コマンドを使用します。

RHEL 9 では、バージョンを指定しない **python** コマンドは、デフォルトの **Python 3.9** バージョンを指し、**python3** コマンドおよび **python3.9** コマンドと同等です。

python コマンドは、対話式セッションを対象としています。実稼働環境では、**python3** または **python3.9** を明示的に使用することが推奨されます。

バージョンを指定しない **python** コマンドは、**dnf remove /usr/bin/python** コマンドを使用してアンインストールできます。

別の python コマンドが必要な場合は、**/usr/local/bin**、**~/.local/bin**、または Python 仮想環境でカスタムシンボリックリンクを作成できます。

バージョンを指定しないコマンド (**python3-pip** パッケージの **/usr/bin/pip** など) がいくつか利用できます。RHEL 9 では、バージョンを指定しないコマンドはすべて、デフォルトの **Python 3.9** バージョンを指します。

アーキテクチャー固有の Python wheels

RHEL 9 に構築されたアーキテクチャー固有の Python **wheels** は、アップストリームアーキテクチャーの命名に準拠しています。これにより、RHEL 9 で Python **wheels** を構築し、RHEL 以外のシステムにインストールできます。以前のリリースの RHEL に構築された Python **wheels** は、上位互換性があ

り、RHEL 9 にインストールできます。これは、Python 拡張機能を含む**wheels** (アーキテクチャーごとに構築) にのみ影響を及ぼし、純粋な Python コードの Python **wheels** (アーキテクチャー固有ではない) には影響を及ぼさない点に注意してください。

第2章 PYTHON のインストールおよび使用

RHEL 9 では、**Python 3.9** はデフォルトの **Python** 実装になります。バージョンを指定しない **python** コマンドは、デフォルトの **Python 3.9** バージョンを指します。

2.1. PYTHON 3 のインストール

デフォルトの Python 実装は通常、デフォルトでインストールされます。手動でインストールするには、以下の手順に従います。

手順

- Python をインストールするには、以下を使用します。

```
# dnf install python3
```

検証手順

- システムにインストールされている Python バージョンを確認するには、以下のコマンドを使用します。

```
$ python3 --version
```

2.2. PYTHON 3 追加パッケージのインストール

python3 で始まるパッケージには、デフォルトの **Python 3.9** バージョンのモジュールが含まれています。

手順

- Python 2 の **Requests** モジュールをインストールするには、以下を使用します。

```
# dnf install python3-requests
```

- Python から **pip** パッケージインストーラーをインストールするには、以下を使用します。

```
# dnf install python3-pip
```

2.3. 開発者用の PYTHON 3 追加ツールのインストール

開発者用の追加の Python ツールは、CodeReady Linux Builder リポジトリーで配布されます。

このリポジトリーには、**python3-pytest** パッケージ、**python3-Cython** パッケージなどが含まれています。

重要

CodeReady Linux Builder リポジトリーおよびそのコンテンツは、Red Hat ではサポートされていません。

リポジトリーからパッケージをインストールするには、以下の手順を行います。

手順

1. CodeReady Linux Builder リポジトリーを有効にします。

```
# subscription-manager repos --enable codeready-builder-for-rhel-9-x86_64-rpms
```

2. **python3-pytest** パッケージをインストールします。

```
# dnf install python3-pytest
```

関連情報

- How to enable and make use of content within CodeReady Linux Builder

2.4. PYTHON の使用

以下の手順では、Python インタープリターまたは Python 関連のコマンドを実行する例を示します。

前提条件

- Python がインストールされていることを確認します。

手順

- Python インタープリターまたは関連コマンドを実行するには、以下を使用します。

```
$ python3
$ python3 -m pip --help
$ python3 -m pip install package
```

第3章 PYTHON 3 RPM のパッケージ化

Python パッケージは、**pip** インストーラーを使用してアップストリームの PyPI リポジトリーから、または DNF パッケージマネージャーを使用してシステムにインストールできます。DNF は RPM パッケージ形式を使用します。これにより、ソフトウェアのダウンストリーム制御が強化されます。

ネイティブ Python パッケージのパッケージ形式は、Python Packaging Authority (PyPA) 仕様 によって定義されています。ほとんどの Python プロジェクトは、パッケージ化に **distutils** または **setuptools** ユーティリティーを使用し、**setup.py** ファイルでパッケージ情報を定義します。ただし、ネイティブ Python パッケージを作成する可能性は時間とともに進化してきました。新しいパッケージング標準の詳細は、pyproject-rpm-macros を参照してください。

この章では、**setup.py** を使用する Python プロジェクトを RPM パッケージにパッケージ化する方法を説明します。このアプローチには、ネイティブ Python パッケージと比較して次の利点があります。

- Python および Python 以外のパッケージへの依存は可能であり、**DNF** パッケージマネージャーによって厳密に適用されます。
- パッケージに暗号で署名できます。暗号化署名を使用すると、RPM パッケージのコンテンツを他のオペレーティングシステムと検証、統合、およびテストできます。
- ビルドプロセス中にテストを実行できます。

3.1. PYTHON パッケージ用の SPEC ファイルの説明

SPEC ファイルには、RPM の構築に **rpmbuild** ユーティリティーを使用する命令が含まれています。命令は、一連のセクションに含まれています。SPEC ファイルには、セクションが定義されている 2 つの主要部分があります。

- プリアンブル (ボディーに使用されている一連のメタデータ項目が含まれています)
- ボディー (命令の主要部分が含まれています)

Python プロジェクトの RPM SPEC ファイルには、非 Python RPM SPEC ファイルと比較していくつかの詳細があります。

重要

Python ライブラリーの RPM パッケージの名前に、**python3** 接頭辞が常に指定されている必要があるということです。

その他の詳細は、次の SPEC ファイルの **python3-pello パッケージの例** に記載されています。その詳細の説明は、例の下に記載されている注意事項を参照してください。

```
Name:           python-pello                                    1
Version:        1.0.2
Release:        1%{?dist}
Summary:        Example Python library

License:        MIT
URL:            https://github.com/fedora-python/Pello
Source:         %{url}/archive/v%{version}/Pello-%{version}.tar.gz

BuildArch:      noarch
```

```
BuildRequires:  python3-devel                                    2

# Build dependencies needed to be specified manually
BuildRequires:  python3-setuptools

# Test dependencies needed to be specified manually
# Also runtime dependencies need to be BuildRequired manually to run tests during build
BuildRequires:  python3-pytest >= 3


%global _description %{expand:
Pello is an example package with an executable that prints Hello World! on the command line.}

%description %_description

%package -n python3-pello                                        3
Summary:        %{summary}

%description -n python3-pello %_description


%prep
%autosetup -p1 -n Pello-%{version}


%build
# The macro only supported projects with setup.py
%py3_build                                                       4


%install
# The macro only supported projects with setup.py
%py3_install


%check                                                           5
%{pytest}


# Note that there is no %%files section for the unversioned python module
%files -n python3-pello
%doc README.md
%license LICENSE.txt
%{_bindir}/pello_greeting

# The library files needed to be listed manually
%{python3_sitelib}/pello/

# The metadata files needed to be listed manually
%{python3_sitelib}/Pello-*.egg-info/
```

1 Python プロジェクトを RPM にパッケージ化するときは、常に **python-** 接頭辞をプロジェクトの元の名前に追加してください。ここでの元の名前は **pello** であるため、**name of the Source RPM (SRPM)** の名前は **python-pello** です。

2 **BuildRequires** は、このパッケージのビルドおよびテストに必要なパッケージを指定します。**BuildRequires** には、Python パッケージのビルドに必要なツールを提供するアイテム

3 バイナリー RPM (ユーザーがインストールできるパッケージ) の名前を選択するときは、バージョン管理された Python 接頭辞 (現在は **python3-**) を追加します。したがって、結果のバイナリー RPM は **python3-pello** という名前になります。

4 **%py3_build** マクロおよび **%py3_install** マクロは、**setup.py build** コマンドおよび **setup.py install** コマンドを実行します。それぞれには、インストール場所、使用するインタープリター、その他の詳細を指定する引数を用います。

5 **%check** セクションは、パッケージ化されたプロジェクトのテストを実行する必要があります。正確なコマンドはプロジェクト自体に大きく依存しますが、**%pytest** マクロを使用して、RPM に適した方法で **pytest** コマンドを実行することができます。**%{python3}** マクロには、Python 3 インタープリターのパス (**/usr/bin/python3**) が含まれています。実際のパスではなく、常にマクロを使用することが推奨されます。

3.2. PYTHON 3 RPM の一般的なマクロ

SPEC ファイルでは、値をハードコーディングするのではなく、以下の **Python 3 RPM のマクロ** の表で説明されているマクロを常に使用します。

表3.1 Python 3 RPM 用のマクロ

マクロ	一般的な定義	説明
%{python3}	/usr/bin/python3	Python3 インタープリター
%{python3_version}	3.9	Python3 インタープリターの major.minor バージョン
%{python3_sitelib}	/usr/lib/python3.9/site-packages	純粋な Python モジュールがインストールされている場所
%{python3_sitearch}	/usr/lib64/python3.9/site-packages	アーキテクチャー固有の拡張モジュールを含むモジュールがインストールされている場所
%py3_build		RPM パッケージに適した引数で **setup.py build** コマンドを実行します。
%py3_install		RPM パッケージに適した引数で **setup.py install** コマンドを実行します。
%{py3_shebang_flags}	s	Python インタープリターディレクティブマクロのデフォルトのフラグセット **%py3_shebang_fix**

マクロ	一般的な定義	説明
%py3_shebang_fix		Python インタープリターディレクティブを **#! %{python3}** に変更すると、既存のフラグ (見つかった場合) を保持し、**%{py3_shebang_flags}** マクロで定義されたフラグを追加します

関連情報

- アップストリームドキュメントの Python マクロ

3.3. PYTHON RPM の自動生成された依存関係の使用

次の手順では、Python プロジェクトを RPM としてパッケージ化するときに自動生成された依存関係を使用する方法を説明します。

前提条件

- RPM の SPEC ファイルが存在します。詳細は、Python パッケージの SPEC ファイルの説明 を参照してください。

手順

1. アップストリームで提供されるメタデータを含む次のディレクトリーのいずれかが、結果の RPM に含まれていることを確認してください。
   - **.dist-info**
   - **.egg-info**
     RPM ビルドプロセスは、これらのディレクトリーから仮想 **pythonX.Ydist** が提供するものを自動的に生成します。次に例を示します。

     ```
     python3.9dist(pello)
     ```

     次に、Python 依存関係ジェネレーターはアップストリームメタデータを読み取り、生成された **pythonX.Ydist** 仮想 provide を使用して各 RPM パッケージのランタイム要件を生成します。たとえば、生成された要件タグは次のようになります。

     ```
     Requires: python3.9dist(requests)
     ```

2. 生成された require を検査します。
3. 生成された require の一部を削除するには、次のいずれかの方法を使用します。
   a. SPEC ファイルの **%prep** セクションでアップストリーム提供のメタデータを変更します。
   b. アップストリームドキュメント で説明されている依存関係の自動フィルタリングを使用します。
4. 自動依存関係ジェネレーターを無効にするには、メインパッケージの **%description** 宣言の上に **%{?python_disable_dependency_generator}** マクロを含めます。

関連情報

- 自動生成された依存関係

第4章 PYTHON スクリプトでインタープリターディレクティブの処理

Red Hat Enterprise Linux 9 では、実行可能な Python スクリプトは、少なくとも主要な Python バージョンを明示的に指定するインタープリターディレクティブ (別名 hashbangs または shebangs) を使用することが想定されます。以下に例を示します。

```
#!/usr/bin/python3
#!/usr/bin/python3.9
```

/usr/lib/rpm/redhat/brp-mangle-shebangs BRP (buildroot policy) スクリプト は、RPM パッケージを構築する際に自動的に実行し、実行可能なすべてのファイルでインタープリターディレクティブを修正しようとします。

BRP スクリプトは、以下のようにあいまいなインタープリターディレクティブを含む Python スクリプトが発生すると、エラーを作成します。

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

または

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

4.1. PYTHON スクリプトでインタープリターディレクティブの変更

次の手順を使用して、RPM ビルド時にビルドエラーが発生する Python スクリプト内のインタープリターディレクティブを変更します。

前提条件

- Python スクリプトのインタープリターディレクティブの一部でビルドエラーが発生します。

手順

- インタープリターディレクティブを変更するには、以下のタスクのいずれかを実行します。
  - SPEC ファイルの **%prep** セクションで次のマクロを使用します。

    ```
    # %py3_shebang_fix SCRIPTNAME …
    ```

    SCRIPTNAME は、任意のファイル、ディレクトリー、またはファイルおよびディレクトリーの一覧にすることができます。

    その結果、一覧になっているすべてのファイルとリストされているディレクトリー内のすべての **.py** ファイルのインタープリターディレクティブは、**%{python3}** を指すように変更されます。元のインタープリターディレクティブの既存のフラグは保持され、**%{py3_shebang_flags}** マクロで定義された追加のフラグが追加されます。SPEC ファイルの **%{py3_shebang_flags}** マクロを再定義して、追加されるフラグを変更できます。

  - **python3-devel** パッケージから **pathfix.py** スクリプトを適用します。

    ```
    # pathfix.py -pn -i %{python3} PATH …
    ```

複数のパスを指定できます。**PATH** がディレクトリーの場合、**pathfix.py** はあいまいなインタープリターディレクティブを持つスクリプトだけでなく、**^[a-zA-Z0-9_]+\.py$** のパターンに一致する Python スクリプトを再帰的にスキャンします。上記のコマンドを **%prep** セクションまたは **%install** セクションの最後に追加します。

- パッケージ化した Python スクリプトを、想定される形式に準拠するように変更します。この目的のために、RPM ビルドプロセスの外部で **pathfix.py** スクリプトを使用することもできます。**pathfix.py** を RPM ビルド以外で実行する場合は、上記の例の **%{python3}** を、**/usr/bin/python3** などのインタープリターディレクティブのパスに置き換えます。

関連情報

- インタープリターの呼び出し

第5章 PHP スクリプト言語の使用

Hypertext Preprocessor (PHP) はサーバー側のスクリプトに主に使用する汎用スクリプト言語で、Web サーバーを使用して PHP コードを実行できるようにします。

5.1. PHP スクリプト言語のインストール

ここでは、パッケージをインストールする方法を説明します。

手順

- PHP をインストールするには、以下を使用します。

```
# dnf install php
```

5.2. WEB サーバーでの PHP スクリプト言語の使用

5.2.1. Apache HTTP Server での PHP の使用

Red Hat Enterprise Linux 9 では、**Apache HTTP Server** で PHP を FastCGI プロセスサーバーとして実行できます。FastCGI Process Manager (FPM) は、Web サイトで高負荷の管理可能にする代替の PHP FastCGI デーモンです。RHEL 9 では、PHP はデフォルトで FastCGI Process Manager を使用します。

本セクションでは、FastCGI プロセスサーバーを使用して PHP コードを実行する方法を説明します。

前提条件

- PHP スクリプト言語がシステムにインストールされている。

手順

1. **httpd** パッケージをインストールします。

```
# dnf install httpd
```

2. **Apache HTTP Server** を起動します。

```
# systemctl start httpd
```

   または、**Apache HTTP Server** をシステムで実行している場合は、PHP のインストール後に **httpd** サービスを再起動します。

```
# systemctl restart httpd
```

3. **php-fpm** サービスを起動します。

```
# systemctl start php-fpm
```

4. オプション:両方のサービスが起動時に開始できるようにします。

```
# systemctl enable php-fpm httpd
```

5. PHP の設定に関する情報を取得するには、以下の内容を含む **index.php** ファイルを **/var/www/html/** ディレクトリーに作成します。

```
echo '<?php phpinfo(); ?>' > /var/www/html/index.php
```

6. **index.php** ファイルを実行するには、ブラウザーで以下を指定します。

```
http://<hostname>/
```

7. オプション:特定の要件がある場合は、設定を調整します。

- **/etc/httpd/conf/httpd.conf** - 一般的な **httpd** 設定
- **/etc/httpd/conf.d/php.conf** - **httpd** の PHP 固有の設定
- **/usr/lib/systemd/system/httpd.service.d/php-fpm.conf** - デフォルトでは、**php-fpm** サービスは **httpd** と一緒に起動します。
- **/etc/php-fpm.conf** - FPM の主要設定
- **/etc/php-fpm.d/www.conf** - デフォルトの **www** プール設定

例5.1「Hello, World」の実行 Apache HTTP Server を使用した PHP スクリプト

1. **/var/www/html/** ディレクトリーにプロジェクト用の **hello** ディレクトリーを作成します。

```
# mkdir hello
```

2. 以下の内容を含む **/var/www/html/hello/** ディレクトリーに **hello.php** ファイルを作成します。

```
# <!DOCTYPE html>
<html>
<head>
<title>Hello, World! Page</title>
</head>
<body>
<?php
    echo 'Hello, World!';
?>
</body>
</html>
```

3. **Apache HTTP Server** を起動します。

```
# systemctl start httpd
```

4. **hello.php** ファイルを実行するには、ブラウザーに以下を指定します。

```
http://<hostname>/hello/hello.php
```

これにより、「Hello, World！」テキストの Web ページが表示されます。

関連情報

- Apache HTTP Web サーバーの設定

5.2.2. nginx Web サーバーでの PHP の使用

本セクションでは、**nginx** Web サーバーで PHP コードを実行する方法を説明します。

前提条件

- PHP スクリプト言語がシステムにインストールされている。

手順

1. **nginx** パッケージをインストールします。

```
# dnf install nginx
```

2. **nginx** サーバーを起動します。

```
# systemctl start nginx
```

または、使用中のシステムで **nginx** サーバーを実行している場合は、PHP のインストール後に **nginx** サービスを再起動します。

```
# systemctl restart nginx
```

3. **php-fpm** サービスを起動します。

```
# systemctl start php-fpm
```

4. オプション:両方のサービスが起動時に開始できるようにします。

```
# systemctl enable php-fpm nginx
```

5. PHP の設定に関する情報を取得するには、以下の内容を含む **index.php** ファイルを **/usr/share/nginx/html/** ディレクトリーに作成します。

```
echo '<?php phpinfo(); ?>' > /usr/share/nginx/html/index.php
```

6. **index.php** ファイルを実行するには、ブラウザーで以下を指定します。

```
http://<hostname>/
```

7. オプション:特定の要件がある場合は、設定を調整します。
   - **/etc/nginx/nginx.conf** - **nginx** main configuration
   - **/etc/nginx/conf.d/php-fpm.conf** - **nginx**の FPM 設定
   - **/etc/php-fpm.conf** - FPM の主要設定
   - **/etc/php-fpm.d/www.conf** - デフォルトの **www** プール設定

例5.2 「Hello, World」の実行 nginx サーバーを使用した PHP スクリプト

1. プロジェクトの **hello** ディレクトリーを **/usr/share/nginx/html/** ディレクトリーに作成します。

```
# mkdir hello
```

2. 以下の内容で **/usr/share/nginx/html/hello/** ディレクトリーに **hello.php** ファイルを作成します。

```
# <!DOCTYPE html>
<html>
<head>
<title>Hello, World! Page</title>
</head>
<body>
<?php
    echo 'Hello, World!';
?>
</body>
</html>
```

3. **nginx** サーバーを起動します。

```
# systemctl start nginx
```

4. **hello.php** ファイルを実行するには、ブラウザーに以下を指定します。

```
http://<hostname>/hello/hello.php
```

これにより、「Hello, World！」テキストの Web ページが表示されます。

関連情報

- NGINX の設定および構成

5.3. コマンドラインインターフェースを使用した PHP スクリプトの実行

通常、PHP スクリプトは Web サーバーを使用して実行されますが、コマンドラインインターフェースを使用して実行することも可能です。

前提条件

- PHP スクリプト言語がシステムにインストールされている。

手順

1. テキストエディターで **filename.php** ファイルを作成します。
filename は、使用するファイル名に置き換えます。

2. コマンドラインから、作成した **filename.php** ファイルを実行します。

```
# php filename.php
```

例5.3 「Hello, World」の実行 コマンドラインインターフェースを使用した PHP スクリプト

1. テキストエディターを使用して、以下の内容で **hello.php** ファイルを作成します。

```
<?php
    echo 'Hello, World!';
?>
```

2. コマンドラインで **hello.php** ファイルを実行します。

```
# php hello.php
```

その結果、「Hello, World！」が出力されます。

5.4. 関連情報

- **httpd(8)** - **httpd** サービスの man ページで、コマンドラインオプションの一覧が記載されます。
- **httpd.conf (5)** - **httpd** 設定ファイルの構造と場所を説明する **httpd** 設定用の man ページです。
- **nginx(8)** - **nginx** Web サーバーの man ページです。コマンドラインオプションの完全一覧とシグナルの一覧が含まれます。
- **php-fpm(8)** - PHP FPM の man ページでは、コマンドラインオプションおよび設定ファイルの全リストが説明されています。