Python Enhancements

Manfred Kaiser

Apr 05, 2022

CONTENTS:

CHAPTER

ONE

INSTALLATION

CHAPTER

TWO

MODULEPARSER

Das Enhancements-Package bietet ein Modulsystem, das sehr einfach in Python Anwendungen integriert werden kann.

Durch das Modulsystem ist es möglich über Kommandozeilenparameter eigene Module zu laden.

Dieses Modulsystem kann darüberhinaus mit Plugins erweitert werden, um zusätzliche Funktionen hinzuzufügen.

Ebenso ist es möglich über den ModuleParser Kommandozeilenparameter anzugeben. Hierfür können die gleichen Parameter wie bei argparse verwendet werden.

2.1 Unterschied Modul und Plugin

Der ModuleParser kann Module und Plugins laden. Der Unterschied zwischen Modulen und Plugins ist, dass Plugins die Funktionalität des ModuleParsers erweitert und Module die Applikation, die den ModuleParser einbindet.

2.2 Erstellen eines ModuleParser

Folgendes Beispiel erstellt erstellt einen ModuleParser und fügt Kommandoozeilenargumente hinzu.

In diesem Beispiel soll ein einfacher CLI Client erstellt werden, über den es möglich ist HTTP Anfragen zu senden.

```
# -*- coding: utf-8 -*-

import requests
from enhancements.modules import ModuleParser

parser = ModuleParser(description='Simple HTTP Client')

parser.add_argument(
    'method',
    action='store',
    choices=['get', 'post'],
    help='HTTP Method'
)

parser.add_argument(
    'url',
    action='store',
    help='URL to open'
)
```

(continues on next page)

(continued from previous page)

```
args = parser.parse_args()

if args.method == 'get':
    response = requests.get(args.url)
else:
    response = requests.post(args.url)

print("Status: {}".format(response.status_code))
```

Dieses Beispiel unterscheidet sich, bis auf die Verwendung des ModuleParsers nicht von einem Programm das den ArgumentParser aus dem argparse-Module verwendet.

2.3 Plugins des ModuleParsers

Derzeit gibt es zwei Plugins für den ModuleParser. Mit diesen ist es möglich ein Logging zur Applikation hinzuzufügen und Standardkonfigurationsdateien zu verwenden.

Die Konfiguration eines Plugins erfolgt über Klassen Eigenschaften. Um ein Plugin zu laden wird die Klasse übergeben.

2.3.1 Logging-Plugin

Das Config Plugin kann über die Klasse `enhancements.plugins.LogModule` eingebunden werden.

Das Logging Plugin konfiguriert das Python Logging Module, so dass Meldungen ab Info angezeigt werden. Um auch Debug Meldungen zu sehen wird ein Kommandozeilenparameter `-d` bzw. `--debug` hinzugefügt.

Ebenso wird ein Parameter `--logfile` hinzugefügt, mit dem es möglich ist eine Datei anzugeben, in die das Log geschrieben werden soll.

Das Logging-Plugin kann so konfiguriert werden, das standardmäßig immer eine Logdatei geschrieben wird. In diesem Fall kann der Parameter `--logfile` dazu verwendet werden um das Log in eine andere Datei zu schreiben. Für den Fall das keine Logdatei erstellt werden soll kann der Parameter `--no-logfile` verwendet werden.

```
# -*- coding: utf-8 -*-

import logging
from enhancements.modules import ModuleParser
from enhancements.plugins import LogModule

parser = ModuleParser(description='Logging Example')

# optionale Konfiguration der Logdatei
LogModule.LOGFILE = '/var/log/example.log'

parser.add_plugin(LogModule)

args = parser.parse_args()

logging.debug("Das ist eine Debug Meldung")
logging.info("Das ist eine Info Meldung")
```

2.3.2 Config-Plugin

Das Config Plugin kann über die Klasse `enhancements.plugins.ConfigModule` eingebunden werden.

Mit diesem Plugin ist es möglich Konfigurationsdateien für Applikationen zu verwalten. Dieses Plugin basiert auf dem Python ConfigParser Modul, erweitert dieses jedoch um die Möglichkeit eine Standardkonfiguration im Package der Applikation zu hinterlegen.

Darüberhinaus ist es möglich Module über die Konfigurationsdatei zu laden.

Auf die Konfigurationsdatei kann über die geparsten Kommandozeilenargumente über `.config` zugegriffen werden. Hierbei ist das ConfigParser Objekt direkt verfügbar.

Note: Um folgendes Beispiel zu testen, erstellen Sie ein neues Package. Das Config-Plugin ist nicht dafür gedacht ausserhalb eines Packages verwendet zu werden.

Erstellen Sie in Ihrem Package ein Konfiguration in die Datei `data/default.ini`.

```
[productionconfig]
configpath = /etc/appname/production.ini

[network]
ip = 192.168.0.1
```

In der Datei `cli.py` fügen Sie folgenden Code ein:

```
# -*- coding: utf-8 -*-

from enhancements.modules import ModuleParser
from enhancements.plugins import ConfigModule

def main():
    parser = ModuleParser(description='Config Example')

    parser.add_plugin(ConfigModule)

    args = parser.parse_args()

    print(args.config.get('network', 'ip'))
```

Nachdem das Package erstellt wurde, können Sie dieses installieren und das entsprechende CLI Tool ausführen.

In der Konfigurationsdatei des Packages wurd eine Bereich mit dem Namen `[productionconfig]` und dem Schlüssel `configpath` definiert. Diese Konfiguration ist optional. Wird diese angegeben, wird geprüft, ob diese Datei existiert und geladen.

Note: In der Production-Konfigurationsdatei müssen nur die Werte angegeben werden, die sich von der Standard-Konfiguratinsdatei des Packages unterscheiden.

CHAPTER

THREE

MODUL ENTWICKLUNG

Mit dem ModuleParser ist es möglich Module zu laden, mit denen es möglich ist bestehende Appliaktionen zu erweitern.

Standardmäßig bieter der ModuleParser diese Möglichkeit nicht an. Um Module laden zu können, muss der ModuleParser entsprechend initialisiert werden.

Note: Es ist empfehlenswert, Die Klasse `Module` nicht direkt zu verwenden, sondern eine eigene Basisklasse für Module zu erstellen, von der dann alle Module erben können. Auf diese Weise kann man besser kontrollieren, welche Module geladen werden dürfen.

```
class MyModule(Module):

    def execute(self, data):
        raise NotImplementedError("execute method must be implemented")
```

Diese Klasse kann als Basisklasse beim ModuleParser angegeben werden. Dadurch ist es möglich die Module auf bestimmte Modultypen einzuschränken. Um mehr als eine Basisklasse zu erlauben ist es möglich ein Tuple von Basisklassen anzugeben. Hierdurch ist es möglich Module unterschiedlicher Typen zu laden.

```
from enhancements.modules import Module, ModuleParser
from mypkg.modules import MyModule

def main():
    parser = ModuleParser(baseclass=MyModule, description='Module Example')
```

Neben der definition der Basisklassen kann auch ein Standardmodul geladen werden, das immer geladen werden soll.

Note: Es ist zu beachten, dass dieses Modul immer geladen wird. Auch wenn andere Module geladen werden. Soll das Default-Modul ersetzt werden, kann dies mit dem Parameter `replace_default=True` angegeben werden.

Anschließend ist es möglich beim Programmstart Module zu übergeben. Diese können mit dem Parameter `--module` bzw. `-m` angegeben werden.

```
myapp --module Module1
```

Es können auch mehrere Module angegeben werden. Diese werden in der angegebenen Reihenfolge ausgeführt.

```
myapp -m Module1 -m Module2 -m Mdoule3
```

Note: Es ist auch möglich, ein Modul mehrmals zu verwenden.

Module können aus einem PIP-Paket oder einer Python-Datei stammen.

Bei der Verwendung von PIP Paketen kann die Klasse wie unter Python üblich angegeben werden.

```
myapp -m mymodule.MyModule
```

Alternativ ist es auch möglich ein Modul aus einer einzelnen Python-Datei zu verwenden. In diesem Fall kann der absolute bzw. relative Pfad zur Datei angegeben werden, Die Modul-Klasse kann durch einen `:` vom Dateinamen getrennt werden.

```
myapp -m /home/user/function.py:MyModule
```

Warning: Beachten Sie, dass beim Importieren eines Moduls der enthaltene Code ausgeführt wird. Bei Dateien, die als Scripte ausgeführt werden, kann dies dazu führen, dass das Script selber ausgeführt wird, was zu unvorhersehbaren Programmabläufen führen kann!

Stellen Sie sicher, dass die Datei nur als Modul verwendet werden kann, oder falls diese auch als Script ausgeführt werden soll, dass diese folgende Überprüfung für den Script Teil beinhaltet:

```
if __name__ == '__main__':
```

3.1 Entwicklung eigener Module

Module sind Python-Klassen die von `Module` abgeleitet sind.

Es ist ebenfalls möglich den Modulen Kommandozeilenparameter zu übergeben. Diese Parameter können in der Methode `parser_arguments()` angegeben werden.

Note: Die Module können nur auf die Argumente, die in der Methode `parser_arguments` definiert sind, zugreifen. Sollen Argumente verwendet werden, die von einem anderen Argument Parser stammen, verwendet werden, müssen diese in der `__init__` Methode übergeben werden.

Um ein Module zu erstellen ist es notwendig eine Basisklasse zu definieren, die die Schnittstellen des Moduls definiert.

Folgendes Beispiel zeigt ein HexDump Modul, das einen Parameter 'hexwidth' definiert.

```
# -*- coding: utf-8 -*-

import binascii
from enhancements.modules import Module

class ExampleModule(Module):

    def execute(self, data):
        pass

class HexDump(ExampleModule):

```

(continues on next page)

(continued from previous page)

```
    @classmethod
    def parser_arguments(cls):
        cls.parser().add_argument(
            '--hexwidth',
            dest='hexwidth',
            type=int,
            default=16,
            help='width of the hexdump in chars'
        )

    def execute(self, data):
        if isinstance(data, str):
            data = bytes(data, 'UTF-8')
        result = []

        for i in range(0, len(data), self.args.hexwidth):
            s = data[i:i + self.args.hexwidth]
            hexa = list(map(''.join, zip(*[iter(binascii.hexlify(s).decode('utf-8
↪'))]*2)))
            while self.args.hexwidth - len(hexa) > 0:
                hexa.append(' ' * 2)
            text = ''.join([chr(x) if 0x20 <= x < 0x7F else '.' for x in  s])
            addr = '%04X:    %s    %s' % (i, " ".join(hexa), text)
            result.append(addr)

        print('\n'.join(result))
```

Dieses Module kann anschließend in einem eigenem Programm verwendet werden. Folgendes Beispiel stellt ein einfaches Programm dar, mit dem eine Datei als Hex Dump ausgegeben werden kann.

Note: Die Module werden nicht vom ModuleParser initalisiert! Dies muss in der Anwendung selber durchgeführt werden. Am einfachsten kann man die Module folgendermaßen initialisieren:

```
modules = [module() for module in args.modules]
```

```
from enhancements.modules import ModuleParser
from enhancements.examples import ExampleModule

parser = ModuleParser(baseclass=ExampleModule, description='Module Example')
parser.add_argument(
    'file',
)
args = parser.parse_args()

modules = [module() for module in args.modules]

if os.path.isfile(args.file):
    with open(args.file, 'rb') as hexfile:
        data = hexfile.read()
    for module in modules:
```

(continues on next page)

(continued from previous page)

```
        module.execute(data)
else:
    print("File not found")
```

3.2 Module erweitern

Neben dem ModuleParser kann auch ein Modul selber durch weitere Module erweitert werden. Dies bietet den Vorteil, dass dadurch auch Module erweitert werden können und eine Anwendung dadurch sehr modular gestaltet werden kann.

In folgendem Beispiel werden Basisklassen für Main-Module und für Sub-Module definiert um diese besser voneinander trennen zu können.

Anschließend werden 2 Submodule (SubModule1, SubModule2) definiert, die dann dem MainModule1 zugewiesen werden können.

Module können sowohl dem ModuleParser als einem Modul zugewiesen werden. Hierfür wird die Methode "add_module" verwendet.

Note: Es ist zu beachten, dass sich die Module um das initialisieren und ausführen der Module kümmern müssen. Theoretisch ist es auch möglich, dass die Submodule außerhalb eines Module initialisiert werden. Dies ist aber nicht zu empfehlen!

```
from enhancements.modules import Module, ModuleParser


class MainModule(Module):

    def execute(self):
        pass

class SubModule(Module):

    def execute(self):
        print("{} ausgeführt mit Parametern: {}".format(self.__class__, self.args))

class SubModule1(SubModule):

    @classmethod
    def parser_arguments(cls):
        cls.parser().add_argument(
            '--value-1',
            dest='submodule_1_value',
            default=1,
            type=int,
            help='Value for sub module 1'
        )


class SubModule2(SubModule):
    @classmethod
```

(continues on next page)

(continued from previous page)

```
    def parser_arguments(cls):
        cls.parser().add_argument(
            '--value-2',
            dest='submodule_2_value',
            default=2,
            type=int,
            help='Value for sub module 2'
        )


class MainModule1(MainModule):

    @classmethod
    def parser_arguments(cls):
        cls.add_module(
            '--submodule',
            dest='submodule',
            default=SubModule1,
            help='Submodule for main module',
            baseclass=SubModule
        )

    def execute(self):
        print(self.__class__)
        print(self.args)
        self.args.submodule().execute()


def main():
    parser = ModuleParser(baseclass=MainModule, description='Module Example')
    args = parser.parse_args()
    modules = [module() for module in args.modules]
    for m in modules:
        m.execute()
```

3.3 Verwendung von Modulen im Code

Bisher wurde beschrieben, wie Module über die Kommandozeile mit Kommandozeilenparameter konfiguriert werden können.

Es kann aber auch vorkommen, dass ein Modul ohne die Verwendung von Kommandozeilenparametern verwendet werden soll.

Aus diesem Grund ist es möglich, dass man die entsprechenden Parameter beim Initialisieren der Klasse mitgeben kann.

Jeder Kommandozeilenparameter besitzt eine Eigenschaft `dest`. Dieser wird als Name für den Parameter, der beim Initialisieren der Klasse angegeben werden kann, verwendet.

Die Basisklasse für die Module erwartet 3 Parameter:

- args = Kommandozeilenargumente als Array => Standard = None
- namespace = Der Namespace, der für das Parsen verwendet werden soll

- **kwargs = Parameter, die anstelle der Kommandozeilenparameter verwendet werden sollen

Folgendes Beispiel zeigt, wie das `SubModule1` aus dem letzten Beispiel alleine verwendet werden kann:

```
m = SubModule1(submodule_1_value=15)
m.execute()
```

Warning: Bei der Verwenung von Modulen im Code ist darauf zu achten, dass die richtigen Datentypen verwendet werden. Sollte die Eigenschaft `type` gesetzt sein, prüft das Modul, ob der übergebene Wert diesem Datentyp entspricht.

3.4 Zusätzliche Parameter für die __init__-Methode

Die __init__-Methode kann auch um eigene Parameter erweitert werden. In folgendem Beispiel wird die Klasse `SubModule1` um einen zusätzlichen Parameter erweitert. Dieser ist sowohl bei der Verwendung mit Kommandozeilenparametern als auch bei der Verwendung im Code anzugeben.

```
class SubModule1(SubModule):

    def __init__(self, myval, args=None, namespace=None, **kwargs):
        super().__init__(args, namespace, **kwargs)
        print(myval)

    @classmethod
    def parser_arguments(cls):
        cls.parser().add_argument(
            '--value-1',
            dest='submodule_1_value',
            default=1,
            type=int,
            help='Value for sub module 1'
        )

if __name__ == '__main__':
    m = SubModule1(1, submodule_1_value=15)
    m.execute()
```

CHAPTER

FOUR

EXTENDEDCONFIGPARSER

The ExtendedConfigParser is an extended Python Config Parser (https://docs.python.org/3/library/configparser.html) and has the same functions and features.

In addition, the ExtendedConfigParser offers the following additional functions, such as standard configuration files in the package and a production configuration.

4.1 Default configuration file

The ExtendedConfigParser has the possibility to read a standard configuration file from a package. If no package is specified the package, from which the config parser was called is used.

```
from enhancements.config import ExtendedConfigParser

config = ExtendedConfigParser()
```

Alternatively, the `package` parameter can be used to specify a specific Python package to search for a default configuration file.

Likewise, the parameters `productionconfig` and `defaultini` can be used to specify alternate configuration files for the production environment and as default configuration.

Another possibility to specify the production configuration file is to define it in the "default.ini":

```
[productionconfig]
configpath = /etc/appname/production.ini
```

If the specified file exists, it will be loaded. If this file does not exist, a warning is issued.

4.2 Additional methods of the ExtendedConfigParser

The ExtendedConfigParser provides the following methods in addition to all Python Config Parser methods:

4.2.1 copy

With the `copy` method an ExtendedConfigParser can be copied to create independent ConfigParser objects.

4.2.2 append

Auxiliary method for the ModuleParser to load configuration files via command line parameters. `append` can be used as an alternative to `read`.

4.2.3 getlist

With `getlist` it is possible to read a list from a configuration file.

The standard Python config parser does not provide a corresponding method.

4.2.4 getmodule

The `getmodule` method returns a module.

`getmodule` can be applied to both a section and an option.

When used as an option, the class name including path can be added directly to the option.

If a module is to be loaded via a section, this section must have the following entries:

It should be noted that `getmodule` returns a class and not an instance

The instantiation of the class must be done by the application itself.

4.2.5 getplugins

Similar to `getmodules`, `getplugins` can be used to load modules.

However, `getplugins` expects a prefix and returns a list of the modules found.

CHAPTER

FIVE

CONTEXT MANAGER

Context managers in Python can be used to influence the execution of a script.

A detailed documentation of what a context manager is and how to create them can be found at: https://docs.python.org/3/library/contextlib.html

The Enhancmend library contains a collection of context managers that you can use in your programs.

5.1 Memory Limit

With the Memory Limit Context Manager it is possible to influence the execution of a script in such a way, that the code handled by the Contex Manager can be aborted as soon as more memory is required than specified.

```
from enhancements.contextmanager import memorylimit

with memorylimit(1 << 30): # 1 GB
    # read large file or do some other actions
    with open('large_file.csv') as f:
        content = f.readlines()
    return content
return None
```

5.2 ExceptionHandler

With the ExceptionHandler it is possible to store exceptions and process them later.

```
from enhancements.contextmanager import ExceptionHandler

try:
    with ExceptionHandler() as ex_handler:
        raise ValueError()
except ValueError:
    print("cleanup after ValueError")
finally:
    if ex_handler.exception_happened:
        print("raised exception: {}".format(ex_handler.exc_type.__name__))
```

CHAPTER

SIX

RETURNCODE

The ReturnCode class can be used to define ReturnCodes for programs or functions.

This class can be used to ensure that a ReturnCode is only changed if it has a higher priority, than the already stored one.

By default, the new value is only set if the new value is greater than the current value.

6.1 Beispiel

```
from enhancements.returncode import BaseReturnCode

class CustomScanResult(BaseReturnCode):
    class Result(BaseReturnCode.Result):
        pass

    Success = Result('success', 10)
    Skip = Result('skip', 11, skip=True)
    Error = Result('error', 12)
```

The class must inherit from `BaseReturnCode` and have an inner class `Result` that inherits from `BaseReturnCode.Result`.

The results must then use the derived Result class.

Note: If the Result class is missing or the values do not use the derived class, an exception is thrown and the creation of the class is aborted.

The reason for creating a derived Result class is to avoid accidentally using results from another Result class.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- search