SphinxTutorial Documentation

Lyudmil Vladimirov

Jul 08, 2020

Contents:

This document aims to describe a standard way of creating a Python package, which can be documented using Sphinx and then consequently published to ReadTheDocs, so that it's made available to whoever needs to read it.

CHAPTER 1

Installation

First we must make sure we have `Sphinx` installed:

```
$ (sudo) pip install sphinx
```

Also, since this tutorial aims to show how to publish project documentation on ReadTheDocs, then we will also need to download the RTD sphinx theme, as so:

```
$ (sudo) pip install sphinx-rtd-theme
```

For the purposes of this tutorial, the `simpleble` package, which we will be documenting, depends on another package named `bluepy`. In order for `Sphinx` to auto-generate documentation from our comments/docstrings, it must also be able to build our package, and thus we must have any dependencies installed and made visible to `Sphinx`. For this reason, and **ONLY** for this specific tutorial, we need to install `bluepy`, as so:

```
$ (sudo) pip install bluepy
```

Note that as long as we used the same `pip` version to install both `Sphinx` and `bluepy`, then they will both be added to the same `PYTHONPATH`, meaning that `Sphinx` can see `blueby` (at least on our local machine). A common mistake that people do is to, for example, use `pip3` to install one of the two and `pip` for the other, in which case the above requirement will not be satisfied as the two packages are installed for different versions of Python and thus are added to a different `PYTHONPATH`.

CHAPTER 2

Folder structure

Let's start by showcasing the folder structure that we should aim for. Here is the folder structure of an example project, named `simpleble` (See GitHub repo and ReadTheDocs documentation), which is also the package which we will base our tutorial on:

```
simpleble-master
├── docs
│   ├── build
│   ├── make.bat
│   ├── Makefile
│   └── source
├── LICENSE
├── README.md
├── requirements.txt
└── simpleble
    └── simpleble.py
```

In the folder structure above:

- `simpleble-master` is the folder we get when we issue a `git pull/clone` command
- `simpleble-master/docs` is the directory where our Sphinx documentation will reside
- `simpleble-master/docs/build` and `simpleble-master/docs/source` being the Sphinx build and source directories respectively. These folders are autogenerated for us by Sphinx.
- `simpleble-master/simpleble` is the actual Python package directory, where our Python source files reside.

An important note here is that the folder `simpleble-master` is what we will refer to as our *Repository root*, while the folder `simpleble-master/docs` will be our *Sphinx root* or, equivalently, our *Documentation root*. Similarly, `simpleble-master/docs/source` will be our *Sphinx source root* and `simpleble-master/docs/build` is our *Sphinx build root*.

CHAPTER 3

Project setup

3.1 Create a repository and clone it

So we begin by creating a Git repository and adding the `README.md`, `LICENSE` and `.gitignore` files, which are of no importance to this tutorial but are generally standard for Git repos. Now on our local machine we proceed by cloning the repository:

```
youruser@yourpc:~yourWorkspacePath$ (sudo) git clone https://github.com/sglvladi/
↪simpleble
```

Once the cloning is complete, we can check to see what files we have in our repo:

```
youruser@yourpc:~yourWorkspacePath$ cd simpleble-master
youruser@yourpc:~yourWorkspacePath/simpleble-master$ ls
LICENSE  README.md
```

3.2 Include your source codes

Now you can add your package source codes to your repository folder. For this tutorial, we only have a single source file `simpleble.py`, which will need to be placed inside a `simpleble` folder under our *Repository root*.

```
youruser@yourpc:~yourWorkspacePath/simpleble-master$ mkdir simpleble
youruser@yourpc:~yourWorkspacePath/simpleble-master$ cp -R <path-to-source-folder>/
↪simpleble.py ./simpleble
youruser@yourpc:~yourWorkspacePath/simpleble-master$ ls
LICENSE  README.md simpleble
youruser@yourpc:~yourWorkspacePath/simpleble-master$ ls ./simpleble/
simpleble.py
```

CHAPTER 4

Sphinx Quickstart

We begin by creating a Sphinx documentation root directory:

```
youruser@yourpc:~yourWorkspacePath/simpleble-master$ mkdir docs
youruser@yourpc:~yourWorkspacePath/simpleble-master$ cd docs
youruser@yourpc:~yourWorkspacePath/simpleble-master/docs$
```

Next we can call `sphinx-quickstart` to initialise our Sphinx project:

```
youruser@yourpc:~yourWorkspacePath/simpleble-master/docs$ sphinx-quickstart
```

We will be presented with a series of questions, the answer to which can depend from project to project, but a generally safe answer set is presented below:

```
Welcome to the Sphinx 1.7.1 quickstart utility.

Please enter values for the following settings (just press Enter to
accept a default value, if one is given in brackets).

Selected root path: .

You have two options for placing the build directory for Sphinx output.
Either, you use a directory "_build" within the root path, or you separate
"source" and "build" directories within the root path.
> Separate source and build directories (y/n) [n]: y

Inside the root directory, two more directories will be created; "_templates"
for custom HTML templates and "_static" for custom stylesheets and other static
files. You can enter another prefix (such as ".") to replace the underscore.
> Name prefix for templates and static dir [_]: (ENTER)

The project name will occur in several places in the built documentation.
> Project name: simpleble
> Author name(s): Lyudmil Vladimirov
> Project release []: 0.0.1
```

(continues on next page)

(continued from previous page)

```
If the documents are to be written in a language other than English,
you can select a language here by its language code. Sphinx will then
translate text that it generates into that language.

For a list of supported codes, see
http://sphinx-doc.org/config.html#confval-language.
> Project language [en]: (ENTER)

The file name suffix for source files. Commonly, this is either ".txt"
or ".rst".  Only files with this suffix are considered documents.
> Source file suffix [.rst]: (ENTER)

One document is special in that it is considered the top node of the
"contents tree", that is, it is the root of the hierarchical structure
of the documents. Normally, this is "index", but if your "index"
document is a custom template, you can also set this to another filename.
> Name of your master document (without suffix) [index]: (ENTER)

Sphinx can also add configuration for epub output:
> Do you want to use the epub builder (y/n) [n]: (ENTER)
Indicate which of the following Sphinx extensions should be enabled:
> autodoc: automatically insert docstrings from modules (y/n) [n]: y
> doctest: automatically test code snippets in doctest blocks (y/n) [n]: (ENTER)
> intersphinx: link between Sphinx documentation of different projects (y/n) [n]:␣
↪(ENTER)
> todo: write "todo" entries that can be shown or hidden on build (y/n) [n]:
> coverage: checks for documentation coverage (y/n) [n]: (ENTER)
> imgmath: include math, rendered as PNG or SVG images (y/n) [n]: (ENTER)
> mathjax: include math, rendered in the browser by MathJax (y/n) [n]: (ENTER)
> ifconfig: conditional inclusion of content based on config values (y/n) [n]: (ENTER)
> viewcode: include links to the source code of documented Python objects (y/n) [n]: y
> githubpages: create .nojekyll file to publish the document on GitHub pages (y/n)␣
↪[n]: (ENTER)

A Makefile and a Windows command file can be generated for you so that you
only have to run e.g. `make html' instead of invoking sphinx-build
directly.
> Create Makefile? (y/n) [y]: (ENTER)
> Create Windows command file? (y/n) [y]: (ENTER)

Creating file ./source/conf.py.
Creating file ./source/index.rst.
Creating file ./Makefile.
Creating file ./make.bat.

Finished: An initial directory structure has been created.
```

Now we can see that some foldes and files have been autogenerated for us:

```
youruser@yourpc:~yourWorkspacePath/simpleble-master/docs$ ls
build  make.bat  Makefile  source
youruser@yourpc:~yourWorkspacePath/simpleble-master/docs$ cd source
youruser@yourpc:~yourWorkspacePath/simpleble-master/docs$ ls
conf.py  index.rst  _static  _templates
```

Let's see how each of these files matter:

- `source`: This is where all our *.rst* files will reside.
- `source/conf.py`: This is the file where all Sphinx configuration settings (including the settings we specified during the `sphinx-quickstart` setup) are specified. When any `make <builder>` or `sphinx-build <builder>` command is called, Sphinx runs this file to extract the desired configuration.
- `source/index.rst`: This is the file which tells Sphinx how to render our index.html page. In general, each `source/*.rst` file is converted to a corresponding `build/html/*.html` page when `make html` is called.
- *build* : This is the directory where the output of any builder is stored when a `make <builder>` or `sphinx-build <builder>` command is called. For the purposes of this tutorial, we are particularly interested in building html documentation and thus our build outputs will be stored under `build/html`. Note that `build/html` does not exist just yet, but will be autogenerated when we call `make html` later.

CHAPTER 5

Sphinx Configuration

Previously, we saw that `conf.py` is the configuration file that Sphinx calls whenever a `make` command is called. Now it's time we made some configuration changes. Let's locate and edit the `conf.py` file:

```
youruser@yourpc:~yourWorkspacePath/simpleble-master/docs$ cd source
youruser@yourpc:~yourWorkspacePath/simpleble-master/docs/source$ nano conf.py
```

There are two things we need to tweak, while we have `conf.py` open, before we proceed to build our documentation.

5.1 Theme configuration

The first thing we want to do is to configure Sphinx to use the RTD theme we previously installed (see *Installation* step). We do this by finding the line that sets the configuration variable `html_theme` and we modify it as follows:

```
html_theme = 'sphinx_rtd_theme'
```

Note that there exist numerous other nice themes out there, so if you're not particularly interested in publishing to RTD then you can choose (or even create) any theme that suits your needs. We chose `sphinx_rtd_theme` since we do want to publish to RTD, plus it is generally quite a nice and clear theme to read.

5.2 Autodoc configuration

Continuing, if we want Sphinx to autogenerate documentation from the comments of our code using the `autodoc` extension, we have to point Sphinx to the directory in which our Python source codes reside. This can be done by uncommenting and altering the following lines, which are generally found at the top of the file:

```
import os
import sys
sys.path.insert(0, os.path.abspath('../../simpleble/'))
```

Notice how we altered the path specified in line 3 (highlighted). This is because our `conf.py` file is located in `simpleble-master/docs/source`, while our Python source codes, in this case the file `simpleble.py`, are located inside `simpleble-master/simpleble`. This means that when Sphinx is looking for our source codes, it now knows that it must go two directories up from `conf.by` (indicated by the `../../` part of the absolute path) and inside the folder `simpleble-master/simpleble`. This is equivalent to adding the directory of our Python source files to PYTHONPATH.

It is important to note here that the absolute path must be specified in relation to where `conf.py` resides, i.e. our 'Sphinx source root', rather than in relation to the 'Documentation root'.

Next, we proceed by creating a `requirements.txt` file in the root directory of our repository, which lists any dependecies of our package. In our case, `simpleble` depends on a package named `bluepy` which can be generally installed by calling `sudo pip install bluepy`.

Create and open the `requirements.txt` file:

```
youruser@yourpc:~yourWorkspacePath/simpleble-master/docs/source$ cd ../..
youruser@yourpc:~yourWorkspacePath/simpleble-master$ nano requirements.txt
```

Then type the following on the first line of the file:

```
bluepy
```

Save and close the file (Ctrl+X -> Enter -> Y -> Enter).

Apart from specifying the dependencies of our package, the `requirements.txt` file can be used to batch install any such dependencies by calling `sudo pip install -r requirements.txt`. For more information on requirements files, you can have a look at pip's documentation.

We will later also use the `requirements.txt` file when configuring the building process of our ReadTheDocs documentation. If we didn't have the requirements file, then ReadTheDocs would fail to run `autodoc` in order to generate our documentation from the comments in our source code. In other words, when building our documentation, RTD calls a `pip install -r requirements.txt`, which installs any requirements in a virtual environment, which is then where `sphinx-apidoc` is run.

CHAPTER 6

Writing docstrings

There are several different docstring formats which one can use in order to enable Sphinx's `autodoc` extension to automatically generate documentation. For this tutorial we will use the `Sphinx` format, since, as the name suggests, it is the standard format used with Sphinx. Other formats include `Google` (see here) and `NumPy` (see here), but they require the use of Sphinx's `napoleon` extension, which is beyond the scope of this tutorial.

6.1 The Sphinx docstring format

In general, a typical `Sphinx` docstring has the following format:

```
"""[Summary]

:param [ParamName]: [ParamDescription], defaults to [DefaultParamVal]
:type [ParamName]: [ParamType](, optional)
...
:raises [ErrorType]: [ErrorDescription]
...
:return: [ReturnDescription]
:rtype: [ReturnType]
"""
```

A pair of `:param:` and `:type:` directive options must be used for each parameter we wish to document. The `:raises:` option is used to describe any errors that are raised by the code, while the `:return:` and `:rtype:` options are used to describe any values returned by our code. A more thorough explanation of the `Sphinx` docstring format can be found here.

Note that the `...` notation has been used above to indicate repetition and should not be used when generating actual docstrings, as can be seen by the example presented below.

6.2 An example class with docstrings

Let's have a look at a typical class documentation. In this example we show the docstrings written for the `SimpleBleDevice` class, which is defined within our `simpleble` module:

```
class SimpleBleDevice(object):
    """This is a conceptual class representation of a simple BLE device
    (GATT Server). It is essentially an extended combination of the
    :class:`bluepy.btle.Peripheral` and :class:`bluepy.btle.ScanEntry` classes

    :param client: A handle to the :class:`simpleble.SimpleBleClient` client
        object that detected the device
    :type client: class:`simpleble.SimpleBleClient`
    :param addr: Device MAC address, defaults to None
    :type addr: str, optional
    :param addrType: Device address type - one of ADDR_TYPE_PUBLIC or
        ADDR_TYPE_RANDOM, defaults to ADDR_TYPE_PUBLIC
    :type addrType: str, optional
    :param iface: Bluetooth interface number (0 = /dev/hci0) used for the
        connection, defaults to 0
    :type iface: int, optional
    :param data: A list of tuples (adtype, description, value) containing the
        AD type code, human-readable description and value for all available
        advertising data items, defaults to None
    :type data: list, optional
    :param rssi: Received Signal Strength Indication for the last received
        broadcast from the device. This is an integer value measured in dB,
        where 0 dB is the maximum (theoretical) signal strength, and more
        negative numbers indicate a weaker signal, defaults to 0
    :type rssi: int, optional
    :param connectable: `True` if the device supports connections, and `False`
        otherwise (typically used for advertising 'beacons').,
        defaults to `False`
    :type connectable: bool, optional
    :param updateCount: Integer count of the number of advertising packets
        received from the device so far, defaults to 0
    :type updateCount: int, optional
    """

    def __init__(self, client, addr=None, addrType=None, iface=0,
                 data=None, rssi=0, connectable=False, updateCount=0):
        """Constructor method
        """
        super().__init__(deviceAddr=None, addrType=addrType, iface=iface)
        self.addr = addr
        self.addrType = addrType
        self.iface = iface
        self.rssi = rssi
        self.connectable = connectable
        self.updateCount = updateCount
        self.data = data
        self._connected = False
        self._services = []
        self._characteristics = []
        self._client = client

    def getServices(self, uuids=None):
```

(continues on next page)

(continued from previous page)

```
        """Returns a list of :class:`bluepy.blte.Service` objects representing
        the services offered by the device. This will perform Bluetooth service
        discovery if this has not already been done; otherwise it will return a
        cached list of services immediately..

        :param uuids: A list of string service UUIDs to be discovered,
            defaults to None
        :type uuids: list, optional
        :return: A list of the discovered :class:`bluepy.blte.Service` objects,
            which match the provided ``uuids``
        :rtype: list On Python 3.x, this returns a dictionary view object,
            not a list
        """
        self._services = []
        if(uuids is not None):
            for uuid in uuids:
                try:
                    service = self.getServiceByUUID(uuid)
                    self.services.append(service)
                except BTLEException:
                    pass
        else:
            self._services = super().getServices()
        return self._services

    def setNotificationCallback(self, callback):
        """Set the callback function to be executed when the device sends a
        notification to the client.

        :param callback: A function handle of the form
            ``callback(client, characteristic, data)``, where ``client`` is a
            handle to the :class:`simpleble.SimpleBleClient` that invoked the
            callback, ``characteristic`` is the notified
            :class:`bluepy.blte.Characteristic` object and data is a
            `bytearray` containing the updated value. Defaults to None
        :type callback: function, optional
        """
        self.withDelegate(
            SimpleBleNotificationDelegate(
                callback,
                client=self._client
            )
        )

    def getCharacteristics(self, startHnd=1, endHnd=0xFFFF, uuids=None):
        """Returns a list containing :class:`bluepy.btle.Characteristic`
        objects for the peripheral. If no arguments are given, will return all
        characteristics. If startHnd and/or endHnd are given, the list is
        restricted to characteristics whose handles are within the given range.

        :param startHnd: Start index, defaults to 1
        :type startHnd: int, optional
        :param endHnd: End index, defaults to 0xFFFF
        :type endHnd: int, optional
        :param uuids: a list of UUID strings, defaults to None
        :type uuids: list, optional
        :return: List of returned :class:`bluepy.btle.Characteristic` objects
```

(continues on next page)

(continued from previous page)

```
        :rtype: list
        """
        self._characteristics = []
        if(uuids is not None):
            for uuid in uuids:
                try:
                    characteristic = super().getCharacteristics(
                        startHnd, endHnd, uuid)[0]
                    self._characteristics.append(characteristic)
                except BTLEException:
                    pass
        else:
            self._characteristics = super().getCharacteristics(startHnd,
                                                               endHnd)
        return self._characteristics

    def connect(self):
        """Attempts to initiate a connection with the device.

        :return: `True` if connection was successful, `False` otherwise
        :rtype: bool
        """
        try:
            super().connect(self.addr,
                            addrType=self.addrType,
                            iface=self.iface)
        except BTLEException as ex:
            self._connected = False
            return (False, ex)
        self._connected = True
        return True

    def disconnect(self):
        """Drops existing connection to device
        """
        super().disconnect()
        self._connected = False

    def isConnected(self):
        """Checks to see if device is connected

        :return: `True` if connected, `False` otherwise
        :rtype: bool
        """
        return self._connected

    def printInfo(self):
        """Print info about device
        """
        print("Device %s (%s), RSSI=%d dB" %
              (self.addr, self.addrType, self.rssi))
        for (adtype, desc, value) in self.data:
            print("  %s = %s" % (desc, value))
```

Once processed by `autodoc` the generated documentation for the above class looks like this.

6.3 Docstrings in VS code

If you are using VS code, the Python Docstring extension can be used to auto-generate a docstring snippet once a function/class has been written. If you want the extension to generate docstrings in `Sphinx` format, you must set the `"autoDocstring.docstringFormat": "sphinx"` setting, under File > Preferences > Settings.

Note that it is best to write the docstrings once you have fully defined the function/class, as then the extension will generate the full dosctring. If you make any changes to the code once a docstring is generated, you will have to manually go and update the affected docstrings.

CHAPTER 7

Populating our documentation

Now that we have setup everything, it is time we started populating our documentation. First of all, lets have a look at the `index.rst` file that was auto-generated during the *Sphinx Quickstart* step:

```
.. simpleble documentation master file, created by
sphinx-quickstart on Sat Mar 10 15:05:19 2018.
You can adapt this file completely to your liking, but it should at least
contain the root `toctree` directive.

Welcome to simpleble's documentation!
=====================================

.. toctree::
:maxdepth: 2
:caption: Contents:



Indices and tables
==================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

A thorough syntax guide for Restructured Text (reST) and how it is used within Sphinx can be found here. In short, the `=====` underline is used to denote page titles (in this case we have two, since it is our index, but it is good practice to have one title per `.rst` file) and the `.. toctree::` directive is used to generate the directory tree (i.e. the Contents) of our documentation, which for now is empty. Finally, the `:ref:` directives are used to refer to the `genindex.html` , `modindex.html` and `search.html` pages, which are auto-generated by sphinx when we build our documentation.

7.1 Building our documentation

In general, when we want to build our (html) documentation, we do so by calling the following command in our *Documentation root* directory:

```
youruser@yourpc:~yourWorkspacePath/simpleble-master/docs$ make html
```

Running the above command will process any Sphinx source files contained within the `simpleble-master/docs/source` (i.e. our *Sphinx source*) directory and will proceed to generate an `html` folder under the `simpleble-master/docs/build` (i.e. our *Sphinx build*) directory, which contains all the relevant files (e.g. html, css, js etc.) used by a webserver to host our documentation in the form of a website.

It is generally good practice to run a `make clean` command before a `make html` to ensure that our html output is generated from scratch. In some cases, changes to the `toctree` in the `index.rst` file might not result in changes to the navigation sidebar of all pages of your documentation. If this happens, then you should run `make clean` followed by `make html`.

However, if we were to build our documentation at this point, and we viewed the generated `simpleble-master/docs/build/html/index.html` file in a browser, we would notice that it is pretty much empty. This is because we have not added anything to it since it was generated. This is what we'll do next.

7.2 Generating documentation from docstrings

Now let's see how we can auto-generate documentation from the docstrings in our Python source files. The `sphinx-apidoc` command can be used to auto-generate some `.rst` files for our Python module. This can be done as follows:

```
youruser@yourpc:~yourWorkspacePath/simpleble-master/docs$ sphinx-apidoc -o ./source ..
↪/simpleble
Creating file ./source/simpleble.rst.
Creating file ./source/modules.rst.
```

The above command generates two files: `simpleble.rst` and `modules.rst` in our *Sphinx source root* directory. The `modules.rst` file is usefull when more than one modules (`.py` files) are included in our Python source directory, but since we only have one module and to keep thing simple, we can ignore it for the purposes of this tutorial. Let's have a look at the contents of `simpleble.rst`:

```
simpleble module
================

.. automodule:: simpleble
    :members:
    :undoc-members:
    :show-inheritance:
```

Notice that the file does not explicitly contain the generated documentation. Instead, the `.. automodule::` directive is called, with the parameter `simpleble` (i.e. our module) and the directive options `:members:`, `:undoc-members:` and `:show-inheritance:`. This is sufficient for `Sphinx`, or more specifically `autodoc`, to generate a `simpleble.html` file, containing documentation generated from our docstrings, when the `make html` command is called. Check the links for more information on autodoc and the Python domain directives.

7.3 Modifying the files generated by `sphinx-apidoc`

Say, for example, that we wish to change the page title of our module documentation page, as well as split our documentation such as to have a separate sub-title for each of our classes. This can be done in our example by modifying the `simpleble.rst` file, as follows:

```
Documentation
=============

The ``SimpleBleClient`` class
*****************************
.. autoclass:: simpleble.SimpleBleClient
    :members:
    :undoc-members:
    :show-inheritance:


The ``SimpleBleDevice`` class
*****************************
.. autoclass:: simpleble.SimpleBleDevice
    :members:
    :undoc-members:
    :show-inheritance:


The ``SimpleBleScanDelegate`` class
***********************************
.. autoclass:: simpleble.SimpleBleScanDelegate
    :members:
    :undoc-members:
    :show-inheritance:

The ``SimpleBleNotificationDelegate`` class
*******************************************
.. autoclass:: simpleble.SimpleBleNotificationDelegate
    :members:
    :undoc-members:
    :show-inheritance:
```

Notice that we have used the `.. autoclass::` directive instead of `.. automodule::`, which in essence instructs `autodoc` to only generate the documentation for the class identified by the classname specified after it. Note, however, that we must prefix the names of our classes with the name of the module. This will ensure that there are not name conflicts, especially when creating documentation for a number of modules.

7.4 Adding pages to our documentation

Even though we have generated the `simpleble.rst` file, in order to add it to our documentation we still need to reference it within the `toctree` specified in our `index.rst` file.

While we are here, let's also create and add an "Introduction" and "Examples" page to our documentation. We start by creating a `intro.rst` file in our *Sphinx source root* (the same directory as the `index.rst` and `simpleble.rts` files), with the following content:

```
Introduction
============
```

(continues on next page)

(continued from previous page)

```
``simpleble`` is a high-level OO Python package which aims to provide an easy and␣
↪intuitive way of interacting with nearby Bluetooth Low Energy (BLE) devices (GATT␣
↪servers). In essence, this package is an extension of the ``bluepy`` package␣
↪created by Ian Harvey (see `here <https://github.com/IanHarvey/bluepy/>`_)

The aim here was to define a single object which would allow users to perform the␣
↪various operations performed by the ``bluepy.btle.Peripheral``, ``bluepy.btle.
↪Scanner``, ``bluepy.btle.Service`` and ``bluepy.btle.Characteristic`` classes of␣
↪``bluepy``, from one central place. This functionality is facilitated by the␣
↪``simpleble.SimpleBleClient`` and ``simpleble.SimpleBleDevice`` classes, where the␣
↪latter is an extention/subclass of ``bluepy.btle.Peripheral``, combined with␣
↪properties of ``bluepy.btle.ScanEntry``.

The current implementation has been developed in Python 3 and tested on a Raspberry␣
↪Pi Zero W, running Raspbian 9 (stretch), but should work with Python 2.7+ (maybe␣
↪with minor modifications in terms of printing and error handling) and most Debian␣
↪based OSs.

Motivation
**********

As a newbie experimenter/hobbyist in the field of IoT using BLE communications, I␣
↪found it pretty hard to identify a Python package which would enable one to use a␣
↪Raspberry Pi (Zero W inthis case) to swiftly scan, connect to and read/write from/
↪to a nearby BLE device (GATT server).

This package is intended to provide a quick, as well as (hopefully) easy to undestand,
↪ way of getting a simple BLE GATT client up and running, for all those out there,␣
↪who, like myself, are hands-on learners and are eager to get their hands dirty from␣
↪early on.

Limitations
***********

- As my main use-case scenario was to simply connect two devices, the current version␣
↪of :class:`simpleble.SimpleBleClient` has been designed and implemented with this␣
↪use-case in mind. As such, if you are looking for a package to allow you to connect␣
↪to multiple devices, then know that off-the-self this package DOES NOT allow you to␣
↪do so. However, implementing such a feature is an easily achievable task, which has␣
↪been planned for sometime in the near future and if there proves to be interest on␣
↪the project, I would be happy to speed up the process.

- Only Read and Write operations are currently supported, but I am planning on adding␣
↪Notifications soon.

- Although the interfacing operations of the :class:`bluepy.btle.Service` and␣
↪:class:`bluepy.btle.Peripheral` classes have been brought forward to the␣
↪:class:`simpleble.SimpleBleClient` class, the same has not been done for the␣
↪:class:`bluepy.btle.Descriptor`, meaning that the :class:`simpleble.
↪SimpleBleClient` cannot be used to directly access the Descriptors. This can␣
↪however be done easily by obtaining a handle of a :class:`simpleble.
↪SimpleBleDevice` object and calling the superclass :meth:`bluepy.btle.Peripheral.
↪getDescriptors` method.
```

There are a few other things that we can observe in the `index.rst`. First is the usage of the `****` underline to specify sub-titles (or sub-sections) in our "Introduction" page. These will be shown as sub-entries to the `toctree`

(Contents) section of our `index.html` file, once built. What is more, we can see how we can use the `:class:` and `:meth:` directive options, when we wish to refer to specific classed and methods. In case of classes and methods defined within our modules, the `:class::` and `:meth:` directives also create clickable links to the corresponding classes and methods of our documentation.

Next, we create a `examples.rst` file in the same directory, which contains a working example, which shows how our `simpleble` module can be used:

```
Examples
=============

Installation/Usage:
*******************
As the package has not been published on PyPi yet, it CANNOT be install using pip.

For now, the suggested method is to put the file `simpleble.py` in the same directory␣
↪as your source files and call ``from simpleble import SimpleBleClient,␣
↪SimpleBleDevice``.

``bluepy`` must also be installed and imported as shown in the example below.
For instructions about how to install, as well as the full documentation of,␣
↪``bluepy`` please refer `here <https://github.com/IanHarvey/bluepy/>`_

Search for device, connect and read characteristic
**************************************************
.. code-block:: python

    """This example demonstrates a simple BLE client that scans for devices,
    connects to a device (GATT server) of choice and continuously reads a␣
↪characteristic on that device.

    The GATT Server in this example runs on an ESP32 with Arduino. For the
    exact script used for this example see `here <https://github.com/nkolban/ESP32_
↪BLE_Arduino/blob/6bad7b42a96f0aa493323ef4821a8efb0e8815f2/examples/BLE_notify/BLE_
↪notify.ino/>`_
    """

    from bluepy.btle import *
    from simpleble import SimpleBleClient, SimpleBleDevice

    # The UUID of the characteristic we want to read and the name of the device # we␣
↪want to read it from
    Characteristic_UUID = "beb5483e-36e1-4688-b7f5-ea07361b26a8"
    Device_Name = "MyESP32"

    # Define our scan and notification callback methods
    def myScanCallback(client, device, isNewDevice, isNewData):
        client._yes = True
        print("#MAC: " + device.addr + " #isNewDevice: " +
            str(isNewDevice) + " #isNewData: " + str(isNewData))
    # TODO: NOTIFICATIONS ARE NOT SUPPORTED YET
    # def myNotificationCallback(client, characteristic, data):
    #     print("Notification received!")
    #     print("  Characteristic UUID: " + characteristic.uuid)
    #     print("  Data: " + str(data))

    # Instantiate a SimpleBleClient and set it's scan callback
    bleClient = SimpleBleClient()
```

(continues on next page)

(continued from previous page)

```
    bleClient.setScanCallback(myScanCallback)
    # TODO: NOTIFICATIONS ARE NOT SUPPORTED YET
    # bleClient.setNotificationCallback(myNotificationCollback)

    # Error handling to detect Keyboard interrupt (Ctrl+C)
    # Loop to ensure we can survive connection drops
    while(not bleClient.isConnected()):
        try:
            # Search for 2 seconds and return a device of interest if found.
            # Internally this makes a call to bleClient.scan(timeout), thus
            # triggering the scan callback method when nearby devices are detected
            device = bleClient.searchDevice(name="MyESP32", timeout=2)
            if(device is not None):
                # If the device was found print out it's info
                print("Found device!!")
                device.printInfo()

                # Proceed to connect to the device
                print("Proceeding to connect....")
                if(bleClient.connect(device)):

                    # Have a peek at the services provided by the device
                    services = device.getServices()
                    for service in services:
                        print("Service ["+str(service.uuid)+"]")

                    # Check to see if the device provides a characteristic with the
                    # desired UUID
                    counter = bleClient.getCharacteristics(
                        uuids=[Characteristic_UUID])[0]
                    if(counter):
                        # If it does, then we proceed to read its value every second
                        while(True):
                            # Error handling ensures that we can survive from
                            # potential connection drops
                            try:
                                # Read the data as bytes and convert to string
                                data_bytes = bleClient.readCharacteristic(
                                    counter)
                                data_str = "".join(map(chr, data_bytes))

                                # Now print the data and wait for a second
                                print("Data: " + data_str)
                                time.sleep(1.0)
                            except BTLEException as e:
                                # If we get disconnected from the device, keep
                                # looping until we have reconnected
                                if(e.code == BTLEException.DISCONNECTED):
                                    bleClient.disconnect()
                                    print(
                                        "Connection to BLE device has been lost!")
                                    break
                                    # while(not bleClient.isConnected()):
                                    #     bleClient.connect(device)

                else:
                    print("Could not connect to device! Retrying in 3 sec...")
```

(continues on next page)

(continued from previous page)

```
                time.sleep(3.0)
            else:
                print("Device not found! Retrying in 3 sec...")
                time.sleep(3.0)
        except BTLEException as e:
            # If we get disconnected from the device, keep
            # looping until we have reconnected
            if(e.code == BTLEException.DISCONNECTED):
                bleClient.disconnect()
                print(
                    "Connection to BLE device has been lost!")
                break
        except KeyboardInterrupt as e:
            # Detect keyboard interrupt and close down
            # bleClient gracefully
            bleClient.disconnect()
            raise e
```

Now that we have generated some `.rst` files, we must add them to our `toctree` by altering the respective part of our `index.rst` file as such:

```
.. simpleble documentation master file, created by
sphinx-quickstart on Sat Mar 10 15:05:19 2018.
You can adapt this file completely to your liking, but it should at least
contain the root `toctree` directive.

Welcome to simpleble's documentation!
=====================================

.. toctree::
:maxdepth: 2
:caption: Contents:

intro
simpleble
examples

Indices and tables
==================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

Notice that it is not necessary to add the `.rst` extension when specifying files in the `toctree`. However, it is **VERY IMPORTANT** that the same indentation with the `..toctree::` directive is maintained, otherwise we will not have the desired output.

7.5 Finalising the documentation

When it is about time to build our documentation for the final time before publishing it, it is always a good idea to run the `make clean` and `make html` pair, to ensure we have a clean build. If we proceed to build the documentation we have generated, we can now see that 3 new entries have been added to our Contents section in `index.html` page, with 4 sub-entries for the "Documentation" page and 2 sub-entries for the "Introduction" and "Examples" pages. When

we click on either of them, we are redirected to a new page, which shows a built representation of the corresponding `*.rst` file.

AND BEHOLD!!! We have generated a Sphinx documentation page for our package! Now it's time to publish it :)

CHAPTER 8

Publishing the documentation to ReadTheDocs

Now that we have finalised and built our documentation, we can procees to upload it to read the docs.

8.1 Pushing our repository

First, we must push our project to the Git host of our choice, as such:

```
youruser@yourpc:~yourWorkspacePath/simpleble-master$ git add .
youruser@yourpc:~yourWorkspacePath/simpleble-master$ git commit
youruser@yourpc:~yourWorkspacePath/simpleble-master$ git push
```

8.2 Importing into ReadTheDocs

Once you have pushed your changes to your Git host, create an account with https://readthedocs.org/ ,if you haven't done so already, and proceed to sign in. Once signed in, there are a number of different ways you can follow to import your project, two of which are described below:

1. Assuming your project is hosted on GitHub, you can proceed by linking your GitHub account to your ReadTheDocs account. By doing so, you can import your project by going on your ReadTheDocs dashboard and clicking the Import a project button. Once there, you will by presented with a list of your GitHub repositories from which you can select the project you want to import by clicking the "+" button next to it's name, in this case `simpleble`. On the next page, simply press the "Next" button and your project will be imported in ReadTheDocs.
2. Alternatively, if you do not wish to link your account, you can follow this link, which should take you to the "Manual Project Import" page. There you should be presented with a page allowing should specify the name, in this case `simpleble`, and the URL, in this case https://github.com/sglvladi/simpleble, of the repository you wish to import. This method has the disadvantage, that a webhook is NOT auto-generated for you when you import the project, meaning that you will have to do this manually (check here), should you wish your documentation to be updated automatically, every time you push to you repository.

8.3 Changing the settings

Once you have followed either of the two approaches of importing your project, outlined above, you should go the the Admin page of your project and under the "Advance Settings" tab, you should specify the path to your `requirements.txt` file, relative to you *Repository root* directory. In the case of *simpleble*, since the `requirements.txt` was placed in the root, you can simply type "requirements.txt".

For Python 3.x projects, such as *simpleble*, you should also make sure to select the CPython 3.x Interpreter at the bottom of the "Advanced Settings" tab. If you don't do so, your project will fail to build.

After you press the "Submit" button, you should be redirected to your project "Overview" page, which should specify the build state of your project. If the page says that "Your documentation is building", simply wait for a few minutes and allow ReadTheDocs to build your project.

8.4 Reading the docs

Once this is done, if everything has been done correctly, your project should pass the building process and you should see a big button on the screen with the text "View your documentation". You can now click this button to view your documentation. Typically, your documentation will be hosted under the domain *yourProjectName.readthedocs.io*. In the case of the "simpleble", the built documentation can be found under simpleble.readthedocs.io.

Well, I hope you enjoyed the tutorial and that it helped you at least in some aspect of your Sphinx+RTD learning curve.

Enjoy documenting!

THE END. . . .

CHAPTER 9

Indices and tables

- genindex
- modindex
- search