PyAutoDoc

Generated by Doxygen 1.9.5

1 PyAutoDoc

About

PyAutoDoc is a documentation solution for C++/Python projects. By converting C++ doxygen comment blocks into SWIG docstring feature directives, PyAutoDoc automates documentation transfer between wrapped C++/Python functions and classes.

Installation

As of right now, PyAutoDoc has to be installed from source and requires that cmake, doxygen, and SWIG are installed. PyAutoDoc can be installed as follows.

- Use git to download a copy of PyAutoDoc's source code.
  ```
  git clone https://github.com/amstokely/pyautodoc.git
  ```
- Go into pyautodoc's root directory
  ```
  cd pyautodoc
  ```
- Create a directory to build pyautodoc in.
  ```
  mkdir build
  ```
- Enter the build directory
  ```
  cd build
  ```
- Build pyautodoc with cmake.
  ```
  cmake ..
  make
  make PythonInstall
  ```
- To test the install run
  ```
  python -c "from pyautodoc import AutoDoc"; echo $?
  ```
  which should ouput
  ```
  0
  ```
  if the install was successful

Getting started

After installing PyAutoDoc, define the following doxygen aliases in the project's CMakeLists.txt.

```
[[PythonExample{1}=<b>Python Example</b><br>\code{.py}\1\endcode]]
   [[CppExample{1}=<b>C++ Example</b><br>\code{.cpp}\1\endcode]]
   [[type{1}=<!-- \1 -->]]
   [[function{1}=<!-- \1 -->]]
   [[const=<!--  -->]]
   [[AutoDocIgnore=<!--  -->]]
```

These aliases are used to communicate difficult to parse information to PyAutoDoc, and are hidden from doxygen's HTML, XML, and LaTex. Brief descriptions of each alias are provided below.

- **PyExample{<text>}** <**!--EXAMPLE END--**>
  1. The provided text is a Python usage example.
- **CppExample{<text>}** <**!--EXAMPLE END--**>
  1. The provided text is a C++ usage example.
- **type{<text>}**
  1. The provided text defines the type of a function or parameter
  2. If used with a parameter, place on the line following the parameter's name and/or description.
  3. If used with a function, place next to the @function command.
- **function{<text>}**
  1. The provided text defines the name of a function.
  2. Place on the second of the docstring.
- **const**
  1. Indicates that a function is const.
  2. Place next to the @function or @type command.
- **AutoDocIgnore**
  1. Indicates PyAutoDoc should ignore a class or function.
  2. Place on the first line of the comment block.

Tutorial

In this short tutorial we'll create a light weight C++/Python class called atom, which could be used to store individual atom information that is typically present in a molecular topology file. First create a project root directory.

```
mkdir atom
```

Navigate into the project's root directory

```
cd atom
```

and create a build, src, include, lib, and SWIG directory.

```
mkdir src include lib swig build
```

Copy the following CMakeLists.txt file into the projects root directory

```
cmake_minimum_required(VERSION 3.21)
project(atom LANGUAGES CXX)
file(MAKE_DIRECTORY ${CMAKE_SOURCE_DIR}/lib)
set(SRC_DIR ${CMAKE_SOURCE_DIR}/src)
set(LIB_DIR ${CMAKE_SOURCE_DIR}/lib)
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${CMAKE_SOURCE_DIR}/lib)
find_package (Python3 COMPONENTS Interpreter Development)
find_package(Doxygen REQUIRED)
find_package(SWIG REQUIRED)
```

```
set(SWIG_INTERFACE ${CMAKE_SOURCE_DIR}/swig/atom.i)
include_directories(${CMAKE_SOURCE_DIR}/include)
include_directories(${Python3_INCLUDE_DIRS})
set_source_files_properties(${SWIG_INTERFACE} PROPERTIES CPLUSPLUS ON)
include(${SWIG_USE_FILE})
add_library(_atom SHARED
        src/atom.cpp
        )
set_target_properties(_atom PROPERTIES OUTPUT_DIR ${LIB_DIR} LIBRARY_OUTPUT_DIRECTORY ${LIB_DIR})
set(CMAKE_SWIG_OUTDIR ${CMAKE_LIBRARY_OUTPUT_DIRECTORY})
swig_add_library(
        atom LANGUAGE python
        SOURCES ${SWIG_INTERFACE} ${SRC_DIR}/atom.cpp ${SWIG_INTERFACE}
)
swig_link_libraries(atom _atom)
set(DOXYGEN_PROJECT_NAME Atom)
set(DOXYGEN_ALIASES
        [[PythonExample{1}=<b>Python Example</b><br>\code{.py}\1\endcode]]
        [[CppExample{1}=<b>C++ Example</b><br>\code{.cpp}\1\endcode]]
        [[type{1}=<!-- \1 -->]]
        [[function{1}=<!-- \1 -->]]
        [[const=<!--  -->]]
        [[AutoDocIgnore=<!--  -->]]
        )
set(DOXYGEN_EXTRACT_PRIVATE YES)
set(DOXYGEN_FULL_PATH_NAMES NO)
set(DOXYGEN_OUTPUT_DIRECTORY ${CMAKE_SOURCE_DIR}/docs)
list(APPEND ATOM_DOCUMENTATION_INPUT_FILES ${CMAKE_SOURCE_DIR}/include/atom.h )
doxygen_add_docs(
        atom_docs
        ${ATOM_DOCUMENTATION_INPUT_FILES}
)
set(DOCUMENTATION_ARGS "Doxyfile.atom_docs")
add_custom_target(docs
        COMMAND ${DOXYGEN_EXECUTABLE} ${DOCUMENTATION_ARGS}
        )
```

,the following SWIG input file into the a file named atom.i in the projects SWIG directory

```
%module atom
%include "std_string.i"
%{
#include "atom.h"
%}
%include atom.h
```

,the following header file into the a file named atom.h in the project's include directory

```
#ifndef ATOM_H
#define ATOM_H
#include <string>
#include <vector>
class Atom {
public:
    Atom ();
    Atom (
            int index,
            const std::string &name,
            const std::string &element,
            const std::string &residueName,
            int residueId,
            double mass
    );
    int index () const;
    std::string name ();
    std::string element ();
    std::string residueName ();
    int residueId () const;
    double mass () const;
private:
    int         _index;
    std::string _name;
    std::string _element;
    std::string _residueName;
    int         _residueId;
    double      _mass;
};
#endif //ATOM_H
```

and the following source file into a file named atom.cpp in the project's src directory.

```
#include "../include/atom.h"
Atom::Atom () = default;
Atom::Atom (
        int index,
        const std::string &name,
        const std::string &element,
```

```
        const std::string &residueName,
        int residueId,
        double mass
) {
    this->_index = index;
    this->_name = name;
    this->_element = element;
    this->_residueName = residueName;
    this->_residueId = residueId;
    this->_mass = mass;
}
int Atom::index () const {
    return _index;
}
std::string Atom::name () {
    return _name;
}
std::string Atom::element () {
    return _element;
}
std::string Atom::residueName () {
    return _residueName;
}
int Atom::residueId () const {
    return _residueId;
}
double Atom::mass() const {
    return this->_mass;
}
```

To compile a Python callable version of atom, simply go into the build directory

```
cd build
```

and build with cmake.

```
cmake ..
make
```

Now, navigate into the project's root directory

```
cd ..
```

and create a file called test.py. Copy the below code into test.py.

```
import sys
sys.path.insert(1, 'lib')
from atom import Atom
help(Atom)
```

When you run test.py

```
python test.py
```

it prints the Atom class's docstring.

```
Help on class Atom in atom:
atom.Atom = class Atom(builtins.object)
 |  atom.Atom(*args)
 |
 |  Methods defined here:
 |
 |  __init__(self, *args)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __repr__ = _swig_repr(self)
 |
 |  element(self)
 |
 |  index(self)
 |
 |  mass(self)
 |
 |  name(self)
 |
 |  residueId(self)
 |
 |  residueName(self)
```

Since this Python class is generated from SWIG wrapper code, even if the C++ was documented using normal doxygen, none of documentation would be copied into the Python version. This is where PyAutoDoc comes in. Open the projects header file, atom.h, and add the below doxygen comment block directly above the Atom class declaration.

```
#ifndef ATOM_H
#define ATOM_H
```

```
#include <string>
#include <vector>

/*!
* @class Atom
* @brief Light weight class used to store individual atom information
* typically found in molecular topology files.
*/
class Atom {
public:
    Atom ();
    Atom (
            int index,
            const std::string &name,
            const std::string &element,
            const std::string &residueName,
            int residueId,
            double mass
    );
    int index () const;
    std::string name ();
    std::string element ();
    std::string residueName ();
    int residueId () const;
    double mass () const;
private:
    int         _index;
    std::string _name;
    std::string _element;
    std::string _residueName;
    int         _residueId;
    double      _mass;
};
#endif //ATOM_H
```

Now create a file named generate_documentation.py in the projects SWIG folder and copy the below code into it.

```
from pyautodoc import AutoDoc
import os
atom_docs = AutoDoc(
    '../include/atom.h',
)
if os.path.exists('atom_docs1.i'):
    os.remove('atom_docs1.i')
atom_docs.writeSwigDocString('atom_docs.i')
```

The AutoDoc class parses the documentation content in the provided header file and stores it as a tree of various PyAutoDoc classes. Calling the AutoDoc.writeSwigDocString method writes the AutoDoc tree contents to the provided file in the form of SWIG docstring feature directives. Generate the SWIG docstring interface file by running generate_documentation.py.

```
python generate_documentation.py
```

This creates a file named atom_docs.i, which is the SWIG interface file used to create Python docstrings.

```
%feature("docstring")
Atom
"\nDescription\n"
"-------------------\n"
"    A light weight class for storing individual atom information\n"
"    typically found in molecular topology files."
"\n"
;
```

In order for SWIG to use this file during compilation, you have to add an include directive with the SWIG docstring interface file's path to the primary SWIG interface file.

```
%module atom
%include "std_string.i"
%{
#include "atom.h"
%}
%include "atom_docs.i"
%include atom.h
```

Rebuild the project

```
cd ../build
rm -rf *
cmake ..
make
```

and run test.py.

```
python test.py
```

The terminal output should look like this.

```
Help on class Atom in atom:
atom.Atom = class Atom(builtins.object)
 |  atom.Atom(*args)
 |
 |  Description
 |  -------------------
 |      A light weight class for storing individual atom information
 |      typically found in molecular topology files.
 |
 |  Methods defined here:
 |
 |  __init__(self, *args)
 |      Description
 |      -------------------
 |          A light weight class for storing individual atom information
 |          typically found in molecular topology files.
 |
 |  __repr__ = _swig_repr(self)
 |
 |  element(self)
 |
 |  index(self)
 |
 |  mass(self)
 |
 |  name(self)
 |
 |  residueId(self)
 |
 |  residueName(self)
 |
 |
```

As you can see, the C++ Atom class documentation was completely appended to the wrapped Python version's docstring. Let's add some more documentation to the C++ side. We'll add parameter documentation for each constructor parameter, an in code python example of how to create an Atom class instance, and descriptions for each of the class methods.

```
#ifndef ATOM_H
#define ATOM_H
#include <string>
#include <vector>

/*!
* @class Atom
* @brief A light weight class for storing individual atom information
* typically found in molecular topology files.
*/
class Atom {
public:
    Atom ();

    /*!
* @function{Atom}
*
* @param index The atom's index.
* @type{int}
*
* @param name The atom's name.
* @type{const std::string&}
*
* @param element The atom's element symbol.
* @type{const std::string&}
*
* @param residueName The atom's residue name.
* @type{const std::string&}
*
* @param residueId The atom's residue ID.
* @type{const std::string&}
*
* @param mass The atom's atomic mass in amu.
* @type{double}
*
* @PythonExample{
* from atom import Atom
*
* newAtom = Atom(0, 'CA', 'C', 'GLU', 1, 12.001)
* } <!--EXAMPLE END-->
*/
    Atom (
            int index,
            const std::string &name,
            const std::string &element,
            const std::string &residueName,
            int residueId,
```

```
            double mass
    );

    /*!
    * @function{index} @type{int} @const
    * @brief Return's the atom's index.
    * @return The atom's index.
    */
    int index () const;

    /*!
    * @function{name} @type{std::string}
    * @brief Return's the atom's name.
    * @return The atom's name.
    */
    std::string name ();

    /*!
    * @function{element} @type{std::string}
    * @brief Return's the atom's element symbol.
    * @return The atom's element symbol.
    */
    std::string element ();

    /*!
    * @function{residueName} @type{std::string}
    * @brief Return's the atom's residue name.
    * @return The atom's residue name.
    */
    std::string residueName ();

    /*!
    * @function{residueId} @type{int} @const
    * @brief Return's the atom's residue ID.
    * @return The atom's residue ID.
    */
    int residueId () const;

    /*!
    * @function{mass} @type{double} @const
    * @brief Return's the atom's mass.
    * @return The atom's mass.
    */
    double mass () const;
private:
    int         _index;
    std::string _name;
    std::string _element;
    std::string _residueName;
    int         _residueId;
    double      _mass;
};
#endif //ATOM_H
```

Run generate_documentation.py

```
cd swig
python generate_documentation.py
```

,rebuild the project

```
cd ../build
rm -rf *
cmake ..
make
```

and run test.py.

```
cd ..
python test.py
```

As before, all of the Atom class documentation from the C++ side was copied into the Python version's docstring.

```
Help on class Atom in atom:
atom.Atom = class Atom(builtins.object)
 |  atom.Atom(*args)
 |
 |  Description
 |  --------------------
 |      A light weight class for storing individual atom information
 |      typically found in molecular topology files.
 |
 |  Methods defined here:
 |
 |  __init__(self, *args)
 |      Atom(index, name, element, residueName, residueId, mass)
 |
```

```
|      Parameters
|      ----------
|      index: int
|          The atom's index.
|
|      name: const std::string&
|          The atom's name.
|
|      element: const std::string&
|          The atom's element symbol.
|
|      residueName: const std::string&
|          The atom's residue name.
|
|      residueId: const std::string&
|          The atom's residue ID.
|
|      mass: double
|          The atom's atomic mass in amu.
|
|      Example
|      ----------
|           from atom import Atom
|
|           newAtom = Atom(0, 'CA', 'C', 'GLU', 1, 12.001)
|
|  __repr__ = _swig_repr(self)
|
|  element(self)
|      element()->std::string
|
|      Description
|      ----------
|          Return's the atom's element symbol.
|
|      Returns
|      ----------
|          The atom's element symbol.
|
|  index(self)
|      index()->int
|
|      Description
|      ----------
|          Return's the atom's index.
|
|      Returns
|      ----------
|          The atom's index.
|
|  mass(self)
|      mass()->double
|
|      Description
|      ----------
|          Return's the atom's mass.
|
|      Returns
|      ----------
|          The atom's mass.
|
|  name(self)
|      name()->std::string
|
|      Description
|      ----------
|          Return's the atom's name.
|
|      Returns
|      ----------
|          The atom's name.
|
|  residueId(self)
|      residueId()->int
|
|      Description
|      ----------
|          Return's the atom's residue ID.
|
|      Returns
|      ----------
|          The atom's residue ID.
|
|  residueName(self)
|      residueName()->std::string
|
|      Description
```

```
|      --------------------
|          Return's the atom's residue name.
|
|      Returns
|      --------------------
|          The atom's residue name.
|
```

One thing you might notice is that the types in the Python docstring are C++ types, which might confuse users who are unfamiliar with C++. To address this problem, you can pass a dictionary of C++/Python type substitutions to the Python AutoDoc class constructor. These substitutions are used when generating the SWIG docstring directives. You can define type substitutions for both parameters and functions. This is useful because functions can only return one type in Python, while parameters can take on multiple types. To map all occurrences of std::string to str, all parameter occurrences of double to [np.float64, np.float32, float], and all function occurrences of double to float, modify generate documentation.py as follows.

```
from pyautodoc import AutoDoc
import os
functionCppPyTypes = {
    "double": "float",
    "double&": "float",
    "const double&": "float",
    "std::string": "str",
    "std::string&": "str",
    "const std::string&": "str",
}
parameterCppPyTypes = {
    "double": "[np.float64, float, np.float32]",
    "double&": "[np.float64, float, np.float32]",
    "const double&": "[np.float64, float, np.float32",
    "std::string": "str",
    "std::string&": "str",
    "const std::string&": "str",
}
atom_docs = AutoDoc(
    '../include/atom.h',
    functionCppPyTypes,
    parameterCppPyTypes
)
if os.path.exists('atom_docs.i'):
    os.remove('atom_docs.i')
atom_docs.writeSwigDocString('atom_docs.i')
```

Run generate_documentation.py

```
python generate_documentation.py
```

and rebuild the project and then run test.py.

```
cd ../build
rm -rf *
cmake ..
make
cd ..
python test.py
```

Notice how all C++ types were replaced by the substitutions passed to the AutoDoc Python constructor in generate↩ _documentation.py.

```
Help on class Atom in atom:
atom.Atom = class Atom(builtins.object)
 |  atom.Atom(*args)
 |
 |  Description
 |  --------------------
 |      A light weight class for storing individual atom information
 |      typically found in molecular topology files.
 |
 |  Methods defined here:
 |
 |  __init__(self, *args)
 |      Atom(index, name, element, residueName, residueId, mass)
 |
 |      Parameters
 |      --------------------
 |      index: int
 |          The atom's index.
 |
 |      name: str
 |          The atom's name.
 |
 |      element: str
 |          The atom's element symbol.
 |
```

```
|      residueName: str
|          The atom's residue name.
|
|      residueId: str
|          The atom's residue ID.
|
|      mass: [np.float64, float, np.float32]
|          The atom's atomic mass in amu.
|
|      Example
|      --------------------
|           from atom import Atom
|
|           newAtom = Atom(0, 'CA', 'C', 'GLU', 1, 12.001)
|
|  __repr__ = _swig_repr(self)
|
|  element(self)
|      element()->str
|
|      Description
|      --------------------
|          Return's the atom's element symbol.
|
|      Returns
|      --------------------
|          The atom's element symbol.
|
|  index(self)
|      index()->int
|
|      Description
|      --------------------
|          Return's the atom's index.
|
|      Returns
|      --------------------
|          The atom's index.
|
|  mass(self)
|      mass()->float
|
|      Description
|      --------------------
|          Return's the atom's mass.
|
|      Returns
|      --------------------
|          The atom's mass.
|
|  name(self)
|      name()->str
|
|      Description
|      --------------------
|          Return's the atom's name.
|
|      Returns
|      --------------------
|          The atom's name.
|
|  residueId(self)
|      residueId()->int
|
|      Description
|      --------------------
|          Return's the atom's residue ID.
|
|      Returns
|      --------------------
|          The atom's residue ID.
|
|  residueName(self)
|      residueName()->str
|
|      Description
|      --------------------
|          Return's the atom's residue name.
|
|      Returns
|      --------------------
|          The atom's residue name.
|
```

While Python and C++ are both OOP languages that support polymorphism, they do so to different extents. Unlike C++, Python does not support method overloading Being that method overloading is commonly used in C++, this

poses a significant problem when generating Python interfaces from C++. SWIG addresses this problem by scrambling the names of overloaded C++ methods under the hood, which preserves C++ method overloading on the Python side. Natively, SWIG only copies the docstring for the last overloaded method when generating documentation for a set of overloaded methods. PyAutoDoc addresses this SWIG limitation, by concatenating all docstrings for a set of overloaded methods into one docstring. To illustrate this, lets overload all Atom class methods by defining a setting for each attribute, which will have the same name as the already implemented getter. We'll also add a note, using the doxygen @note special method, to each getter/setter method specifying the method is a getter or setter. This might be helpful for users, since the getters/setters do not have get/set in their names. Additionally, each getter/setter will have a Python example. Replace atom.h with

```
#ifndef ATOM_H
#define ATOM_H
#include <string>
#include <vector>
#include <map>

/*!
* @class Atom
* @brief A light weight class for storing individual atom information
* typically found in molecular topology files.
*/
class Atom {
public:

    /*!
    * @function{Atom}
    *
    * @brief The default constructor. When an instance of Atom is
    * instantiated using this constructor, you have to manually set
    * all of the class attributes.
    *
    *@PythonExample{
    * from atom import Atom
    *
    * newAtom = Atom()
    * newAtom.index(0)
    * newAtom.name('CA')
    * newAtom.element("C")
    * newAtom.residueName("GLU")
    * newAtom.residueId(1)
    * newAtom.mass(12.001)
    *
    * }<!--EXAMPLE END-->
    */
    Atom ();

    /*!
* @function{Atom}
*
* @param index The atom's index.
* @type{int}
*
* @param name The atom's name.
* @type{const std::string&}
*
* @param element The atom's element symbol.
* @type{const std::string&}
*
* @param residueName The atom's residue name.
* @type{const std::string&}
*
* @param residueId The atom's residue ID.
* @type{const std::string&}
*
* @param mass The atom's atomic mass in amu.
* @type{double}
*
* @PythonExample{
* from atom import Atom
*
* newAtom = Atom(0, 'CA', 'C', 'GLU', 1, 12.001)
* } <!--EXAMPLE END-->
*/
    Atom (
            int index,
            const std::string &name,
            const std::string &element,
            const std::string &residueName,
            int residueId,
            double mass
    );

    /*!
    * @function{index} @type{int} @const
```

```
* @brief Return's the atom's index.
* @return The atom's index.
* @note This method is the index attribute getter
*
* } <!--EXAMPLE END-->
*/
int index () const;

/*!
* @function{index} @type{void}
*
* @param index The atom's index.
* @type{int}
*
* @brief Sets the atom's index.
* @note This method is the index attribute setter.
*
* @PythonExample{
*
* from atom import Atom
*
* newAtom = Atom()
* newAtom.index(0)
*
* index = newAtom.index()
* print(index)
*
* '''
* --> 0
* '''
* } <!--EXAMPLE END-->
*/
void index (int index);

/*!
* @function{name} @type{std::string}
* @brief Return's the atom's name.
* @return The atom's name.
* @note This method is the name attribute getter.
*/
std::string name ();

/*!
* @function{name} @type{void}
* @param name The atom's name.
* @type{std::string}
*
* @brief Sets the atom's name.
* @note This method is the name attribute setter.
*
* @PythonExample{
*
* from atom import Atom
*
* newAtom = Atom()
* newAtom.name("CA)
*
* name = newAtom.name()
* print(name)
*
* '''
* --> CA
* '''
* } <!--EXAMPLE END-->
*/
void name (std::string name);

/*!
* @function{element} @type{std::string}
* @brief Return's the atom's element symbol.
* @return The atom's element symbol.
* @note This method is the element attribute getter.
*
*/
std::string element ();

/*!
* @function{element} @type{void}
* @param element The atom's element symbol.
* @type{std::string}
*
* @brief Sets the atom's element symbol.
* @note This method is the element attribute setter.
*
* @PythonExample{
*
* from atom import Atom
```

```
*
* newAtom = Atom()
* newAtom.element("C")
*
* element = newAtom.element()
* print(element)
*
* '''
* --> C
* '''
*
* } <!--EXAMPLE END-->
*
*/
void element (std::string element);

/*!
* @function{residueName} @type{std::string}
* @brief Return's the atom's residue name.
* @return The atom's residue name.
* @note This method is the residueName attribute getter.
*/
std::string residueName ();

/*!
* @function{residueName} @type{void}
* @param residueName The atom's residue name.
* @type{std::string}
*
* @brief Sets the atom's residue name.
* @note This method is the residueName attribute setter.
*
* @PythonExample{
*
* from atom import Atom
*
* newAtom = Atom()
* newAtom.residueName("GLU")
*
* residueName = newAtom.residueName()
* print(residueName)
*
* '''
* --> GLU
* '''
* } <!--EXAMPLE END-->
*/
void residueName (std::string residueName);

/*!
* @function{residueId} @type{int} @const
* @brief Return's the atom's residue ID.
* @return The atom's residue ID.
* @note This method is the residueId attribute getter.
*/
int residueId () const;

/*!
* @function{residueId} @type{void}
* @param residueId The atom's residue ID.
* @type{int}
*
* @brief Sets the atom's residue ID.
* @note This method is the residueId attribute setter.
*
* @PythonExample{
*
* from atom import Atom
*
* newAtom = Atom()
* newAtom.residueId(1)
*
* residueId = newAtom.residueId()
* print(residueId)
*
* '''
* --> 1
* '''
* } <!--EXAMPLE END-->
*/
void residueId (int residueId);

/*!
* @function{mass} @type{double} @const
* @brief Return's the atom's mass.
* @return The atom's mass.
* @note This method is the mass attribute getter.
```

```
    */
    double mass () const;

    /*!
    * @function{mass} @type{void}
    * @param mass The atom's mass.
    * @type{double}
    *
    * @brief Sets the atom's mass.
    * @note This method is the mass attribute setter.
    *
    * @PythonExample{
    *
    * from atom import Atom
    *
    * newAtom = Atom()
    * newAtom.mass(12.001)
    *
    * mass = newAtom.mass()
    * print(mass)
    *
    * "'
    * --> 12.001
    * "'
    *
    * } <!--EXAMPLE END-->
    */
    void mass (double mass);
private:
    int         _index;
    std::string _name;
    std::string _element;
    std::string _residueName;
    int         _residueId;
    double      _mass;
};
#endif //ATOM_H
```

and atom.cpp with

```
#include "../include/atom.h"
Atom::Atom () = default;
Atom::Atom (
        int index,
        const std::string &name,
        const std::string &element,
        const std::string &residueName,
        int residueId,
        double mass
) {
    this->_index       = index;
    this->_name        = name;
    this->_element     = element;
    this->_residueName = residueName;
    this->_residueId   = residueId;
    this->_mass        = mass;
}
int Atom::index () const {
    return _index;
}
void Atom::index (int index) {
    this->_index = index;
}
std::string Atom::name () {
    return _name;
}
void Atom::name (std::string name) {
    this->_name = name;
}
std::string Atom::element () {
    return _element;
}
void Atom::element (std::string element) {
    this->_element = element;
}
std::string Atom::residueName () {
    return _residueName;
}
void Atom::residueName (std::string residueName) {
    this->_residueName = residueName;
}
int Atom::residueId () const {
    return _residueId;
}
void Atom::residueId (int residueId) {
    this->_residueId = residueId;
}
```

```
double Atom::mass () const {
    return this->_mass;
}
void Atom::mass (double mass) {
    this->_mass = mass;
}
```

Enter the swig directory, open generate_documentation.py and add
`"void" : "None"`

to the functionCppPyTypes dictionary. As before, run generate_documentation.py, rebuild the project, and run test.py.

```
Help on class Atom in atom:
atom.Atom = class Atom(builtins.object)
 |  atom.Atom(*args)
 |
 |  Description
 |  -------------------
 |      A light weight class for storing individual atom information
 |      typically found in molecular topology files.
 |
 |  Methods defined here:
 |
 |  __init__(self, *args)
 |      ********************
 |      *    Version 1     *
 |      ********************
 |      Atom()
 |
 |      Description
 |      --------------------
 |          The default constructor. When an instance of Atom is
 |          instantiated using this constructor, you have to manually set
 |          all of the class attributes.
 |
 |      Example
 |      --------------------
 |           from atom import Atom
 |
 |           newAtom = Atom()
 |           newAtom.index(0)
 |           newAtom.name('CA')
 |           newAtom.element("C")
 |           newAtom.residueName("GLU")
 |           newAtom.residueId(1)
 |           newAtom.mass(12.001)
 |
 |      ********************
 |      *    Version 2     *
 |      ********************
 |      Atom(index, name, element, residueName, residueId, mass)
 |
 |      Parameters
 |      --------------------
 |      index: int
 |          The atom's index.
 |
 |      name: str
 |          The atom's name.
 |
 |      element: str
 |          The atom's element symbol.
 |
 |      residueName: str
 |          The atom's residue name.
 |
 |      residueId: str
 |          The atom's residue ID.
 |
 |      mass: [np.float64, float, np.float32]
 |          The atom's atomic mass in amu.
 |
 |      Example
 |      --------------------
 |           from atom import Atom
 |
 |           newAtom = Atom(0, 'CA', 'C', 'GLU', 1, 12.001)
 |
 |  __repr__ = _swig_repr(self)
 |
 |  element(self, *args)
 |      ********************
 |      *    Version 1     *
 |      ********************
 |      element()->str
```

```
|
|      Description
|      --------------------
|          Return's the atom's element symbol.
|
|      Returns
|      --------------------
|          The atom's element symbol.
|
|
|      Note
|      --------------------
|          This method is the element attribute getter.
|
|      ********************
|      *    Version 2     *
|      ********************
|      element(element)->None
|
|      Description
|      --------------------
|          Sets the atom's element symbol.
|
|      Parameters
|      --------------------
|      element: str
|          The atom's element symbol.
|
|      Note
|      --------------------
|          This method is the element attribute setter.
|
|  index(self, *args)
|      ********************
|      *    Version 1     *
|      ********************
|      index()->int
|
|      Description
|      --------------------
|          Return's the atom's index.
|
|      Returns
|      --------------------
|          The atom's index.
|
|
|      Note
|      --------------------
|          This method is the index attribute getter
|
|          } <!--EXAMPLE END-->
|
|      ********************
|      *    Version 2     *
|      ********************
|      index(index)->None
|
|      Description
|      --------------------
|          Sets the atom's index.
|
|      Parameters
|      --------------------
|      index: int
|          The atom's index.
|
|      Note
|      --------------------
|          This method is the index attribute setter.
|
|      Example
|      --------------------
|
|           from atom import Atom
|
|           newAtom = Atom()
|           newAtom.index(0)
|
|           index = newAtom.index()
|           print(index)
|
|           "'
|           --> 0
|           "'
|
|  mass(self, *args)
```

```
|      ********************
|      *    Version 1     *
|      ********************
|      mass()->float
|
|      Description
|      --------------------
|          Return's the atom's mass.
|
|      Returns
|      --------------------
|          The atom's mass.
|
|
|      Note
|      --------------------
|          This method is the mass attribute getter.
|
|      ********************
|      *    Version 2     *
|      ********************
|      mass(mass)->None
|
|      Description
|      --------------------
|          Sets the atom's mass.
|
|      Parameters
|      --------------------
|      mass: [np.float64, float, np.float32]
|          The atom's mass.
|
|      Note
|      --------------------
|          This method is the mass attribute setter.
|
|      Example
|      --------------------
|
|           from atom import Atom
|
|           newAtom = Atom()
|           newAtom.mass(12.001)
|
|           mass = newAtom.mass()
|           print(mass)
|
|           "'
|           --> 12.001
|           "'
|
|  name(self, *args)
|      ********************
|      *    Version 1     *
|      ********************
|      name()->str
|
|      Description
|      --------------------
|          Return's the atom's name.
|
|      Returns
|      --------------------
|          The atom's name.
|
|
|      Note
|      --------------------
|          This method is the name attribute getter.
|
|      ********************
|      *    Version 2     *
|      ********************
|      name(name)->None
|
|      Description
|      --------------------
|          Sets the atom's name.
|
|      Parameters
|      --------------------
|      name: str
|          The atom's name.
|
|      Note
|      --------------------
|          This method is the name attribute setter.
```

```
|
|      Example
|      --------------------
|
|           from atom import Atom
|
|           newAtom = Atom()
|           newAtom.name("CA)
|
|           name = newAtom.name()
|           print(name)
|
|           "'
|           --> CA
|           "'
|
|  residueId(self, *args)
|      ********************
|      *    Version 1     *
|      ********************
|      residueId()->int
|
|      Description
|      --------------------
|          Return's the atom's residue ID.
|
|      Returns
|      --------------------
|          The atom's residue ID.
|
|
|      Note
|      --------------------
|          This method is the residueId attribute getter.
|
|      ********************
|      *    Version 2     *
|      ********************
|      residueId(residueId)->None
|
|      Description
|      --------------------
|          Sets the atom's residue ID.
|
|      Parameters
|      --------------------
|      residueId: int
|          The atom's residue ID.
|
|      Note
|      --------------------
|          This method is the residueId attribute setter.
|
|      Example
|      --------------------
|
|           from atom import Atom
|
|           newAtom = Atom()
|           newAtom.residueId(1)
|
|           residueId = newAtom.residueId()
|           print(residueId)
|
|           "'
|           --> 1
|           "'
|
|  residueName(self, *args)
|      ********************
|      *    Version 1     *
|      ********************
|      residueName()->str
|
|      Description
|      --------------------
|          Return's the atom's residue name.
|
|      Returns
|      --------------------
|          The atom's residue name.
|
|
|      Note
|      --------------------
|          This method is the residueName attribute getter.
|
```

```
|      Example
|      --------------------
|
|           from atom import Atom
|
|           newAtom = Atom()
|           newAtom.element("C")
|
|           element = newAtom.element()
|           print(element)
|
|           ”’
|           --> C
|           ”’
|
|      ********************
|      *    Version 2     *
|      ********************
|      residueName(residueName)->None
|
|      Description
|      --------------------
|          Sets the atom’s residue name.
|
|      Parameters
|      --------------------
|      residueName: str
|          The atom’s residue name.
|
|      Note
|      --------------------
|          This method is the residueName attribute setter.
|
|      Example
|      --------------------
|
|           from atom import Atom
|
|           newAtom = Atom()
|           newAtom.residueName("GLU")
|
|           residueName = newAtom.residueName()
|           print(residueName)
|
|           ”’
|           --> GLU
|           ”’
|
```

The documentation for all of the overloaded methods was copied into the Python docstrings, and marked with version numbers.
This concludes the tutorial. All final files used in this tutorial can be found in the example/atom PyAutoDoc source code directory.

2 Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

3 Class Documentation

3.1 AutoDoc Class Reference

The user interface class in PyAutoDoc that initiates the transfer of doxygen documentation from C++ to Python. A comprehensive PyAutoDoc tutorial can be found on the documentation main page.

`#include <autodoc.h>`

Public Member Functions

- AutoDoc (const std::string &fname)

  Use this constructor if you want to preserve C++ types in the python documentation.
- AutoDoc (const std::string &fname, std::map< std::string, std::string > *cppPyTypes)

  Use this constructor if you want to replace C++ types with the Python types defined in the cppPyTypes parameter.
- AutoDoc (const std::string &fname, std::map< std::string, std::string > *functionCppPyTypes, std::map< std::string, std::string > *parameterCppPyTypes)

  Use this constructor if you want to replace the C++ types with Python types, but want to use different types for the parameters and functions. This is useful when a wrapped Python function can accept multiple "similar", but can only return one specific type.
- std::map< std::string, std::string > * **functionCppPyTypes** ()
- std::map< std::string, std::string > * **parameterCppPyTypes** ()
- std::vector< AutoDocFunction > **functions** ()
- std::vector< AutoDocClass > **classes** ()
- void writeSwigDocString (const std::string &fname)

  Writes a SWIG docstring feature directive interface file.

Private Attributes

- std::vector< AutoDocFunction > **functions_**
- std::vector< AutoDocClass > **classes_**
- std::map< std::string, std::string > **functionCppPyTypes_**
- std::map< std::string, std::string > **parameterCppPyTypes_**

3.1.1 Detailed Description

The user interface class in PyAutoDoc that initiates the transfer of doxygen documentation from C++ to Python. A comprehensive PyAutoDoc tutorial can be found on the documentation main page.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 AutoDoc() [1/3]

```
AutoDoc::AutoDoc (
        const std::string & fname )  [explicit]
```

Use this constructor if you want to preserve C++ types in the python documentation.

Parameters

fname	The C++ header file name that contains the C++ doxygen documentation

3.1.2.2 AutoDoc() [2/3]

```
AutoDoc::AutoDoc (
        const std::string & fname,
        std::map< std::string, std::string > * cppPyTypes )
```

Use this constructor if you want to replace C++ types with the Python types defined in the cppPyTypes parameter.

Parameters

fname	The C++ header file name that contains the C++ doxygen documentation
cppPyTypes	Map defining the C++/Python type substitutions, where the keys are the C++ types, and the values are the Python types. The same substitutions with be performed for both parameters and functions. In python, cppPyTypes will be a string/string dictionary.

3.1.2.3 AutoDoc() [3/3]

```
AutoDoc::AutoDoc (
        const std::string & fname,
        std::map< std::string, std::string > * functionCppPyTypes,
        std::map< std::string, std::string > * parameterCppPyTypes )
```

Use this constructor if you want to replace the C++ types with Python types, but want to use different types for the parameters and functions. This is useful when a wrapped Python function can accept multiple "similar", but can only return one specific type.

Parameters

fname	The C++ header file name that contains the C++ doxygen documentation
parameterCppPyTypes	Map defining the C++/Python type substitutions used for parameters, where the keys are the C++ types, and the values are the Python types. In python, parameterCppPyTypes will be a string/string dictionary.
functionCppPyTypes	Map defining the C++/Python type substitutions used for functions, where the keys are the C++ types, and the values are the Python types. In python, functionCppPyTypes will be a string/string dictionary.

3.1.3 Member Function Documentation

3.1.3.1 writeSwigDocString()

```
void AutoDoc::writeSwigDocString (
        const std::string & fname )
```

Writes a SWIG docstring feature directive interface file.

Parameters

fname	The name of the output SWIG interface file. The interface file has a ".i" extension and it's path must be in an include directive in the primary SWIG interface file.

The documentation for this class was generated from the following file:

- autodoc.h

Index