

---

**inpRW**

***Release 2023.10.6***

**Erik Kane**

**Nov 21, 2023**



# USER DOCUMENTATION

<b>1</b>	<b>inpRW Utility Introduction</b>	<b>1</b>
1.1	Project Overview . . . . .	1
1.2	Project Goals . . . . .	2
1.3	Important Limitations . . . . .	2
1.4	Projects Which Used inpRW . . . . .	3
1.5	Data Structure . . . . .	3
1.6	Project Structure . . . . .	4
1.7	Documentation Conventions . . . . .	4
<b>2</b>	<b>Project Background</b>	<b>5</b>
2.1	Difficulties of parsing Abaqus input files . . . . .	5
2.2	inpParser Module . . . . .	5
2.3	inpRW History . . . . .	6
<b>3</b>	<b>Installation and Usage Instructions</b>	<b>7</b>
3.1	Option 1: Using install_to_3DEXPERIENCE.bat . . . . .	7
3.2	Option 2: Install inpRW from .whl . . . . .	7
3.3	Option 3: Extract inpRW from .whl . . . . .	8
3.4	Calling inpRW . . . . .	8
3.5	Using inpRW with Keyword Edit in 3DEXPERIENCE . . . . .	8
<b>4</b>	<b>inpRW Tests</b>	<b>9</b>
<b>5</b>	<b>Running inpRW Tests</b>	<b>11</b>
<b>6</b>	<b>inpKeyword Basics</b>	<b>13</b>
6.1	Abaqus Keyword Block Terminology . . . . .	13
6.2	Important inpKeyword Attributes . . . . .	14
6.3	Creating a new inpKeyword object: Recommended option . . . . .	15
6.4	Creating a new inpKeyword object: Alternate option . . . . .	15
6.5	Creating a new inpKeyword object: Hybrid option . . . . .	16
<b>7</b>	<b>Strings, Integers, and Decimals</b>	<b>17</b>
<b>8</b>	<b>Important inpRW Attributes</b>	<b>19</b>
8.1	keywords . . . . .	19
8.2	kwg . . . . .	19
8.3	steps . . . . .	19
8.4	step_paths . . . . .	19
8.5	nd . . . . .	19
8.6	ed . . . . .	20

8.7	<code>inpName</code>	20
8.8	<code>inputFolder</code>	20
8.9	<code>outputFolder</code>	20
8.10	<code>jobSuffix</code>	20
8.11	<code>includeFileNames</code>	20
8.12	<code>includeBlockPaths</code>	20
<b>9</b>	<b>Miscellaneous Usage Hints</b>	<b>21</b>
<b>10</b>	<b>Example Scenarios</b>	<b>23</b>
10.1	Example input file	23
10.2	Modifying data in an existing <code>inpKeyword</code> block	25
10.3	Creating a new keyword block and inserting it into the <code>inp</code> data structure	26
10.4	Deleting a keyword block	27
10.5	Writing the new input file	28
10.6	Complete Python Script	28
10.7	Modified Input File	29
<b>11</b>	<b>Glossary</b>	<b>33</b>
<b>12</b>	<b><code>inpRW</code></b>	<b>35</b>
12.1	Package Contents	35
12.2	Submodules	40
<b>13</b>	<b><code>centroid</code></b>	<b>69</b>
13.1	Module Contents	69
<b>14</b>	<b><code>config</code></b>	<b>71</b>
14.1	Module Contents	71
<b>15</b>	<b><code>config_re</code></b>	<b>75</b>
15.1	Module Contents	75
<b>16</b>	<b><code>csid</code></b>	<b>77</b>
16.1	Module Contents	77
<b>17</b>	<b><code>eval2</code></b>	<b>89</b>
17.1	Module Contents	89
<b>18</b>	<b><code>elType</code></b>	<b>91</b>
18.1	Module Contents	91
<b>19</b>	<b><code>inpDecimal</code></b>	<b>95</b>
19.1	Module Contents	95
<b>20</b>	<b><code>inpInt</code></b>	<b>99</b>
20.1	Module Contents	99
<b>21</b>	<b><code>inpKeyword</code></b>	<b>103</b>
21.1	Module Contents	103
<b>22</b>	<b><code>inpKeywordHelper</code></b>	<b>127</b>
22.1	Module Contents	127
<b>23</b>	<b><code>inpKeywordSequence</code></b>	<b>129</b>
23.1	Module Contents	129

<b>24</b>	<b>inpRWErrors</b>	<b>133</b>
24.1	Module Contents . . . . .	133
<b>25</b>	<b>inpString</b>	<b>137</b>
25.1	Module Contents . . . . .	137
<b>26</b>	<b>mesh</b>	<b>141</b>
26.1	Module Contents . . . . .	141
<b>27</b>	<b>misc_functions</b>	<b>157</b>
27.1	Module Contents . . . . .	157
<b>28</b>	<b>NoneSort</b>	<b>163</b>
28.1	Module Contents . . . . .	163
<b>29</b>	<b>printer</b>	<b>165</b>
29.1	Module Contents . . . . .	165
<b>30</b>	<b>repr2</b>	<b>167</b>
30.1	Module Contents . . . . .	167
<b>31</b>	<b>Update Notes</b>	<b>169</b>
31.1	2023.10.6 . . . . .	169
<b>32</b>	<b>Legal Notices and Usage Disclaimer</b>	<b>177</b>
32.1	inpRW Usage Disclaimer . . . . .	177
32.2	Legal Notices . . . . .	177
	<b>Python Module Index</b>	<b>179</b>
	<b>Index</b>	<b>181</b>



## INPRW UTILITY INTRODUCTION

inpRW is a collection of Python modules for parsing, modifying, and writing *Abaqus* input files. Requires: Python 3.7+, [numpy](#), [scipy](#)

### 1.1 Project Overview

- The module will parse the data in an Abaqus input file to Python objects, which allows a script to recognize values in an input file.
  - For example, keyword blocks, comments, and data lines are treated separately rather than as a large block of text.
- There are several functions built in to modify the input file data. Here are some examples of functions built-in to inpRW:
  - Reorganize step data so multiple nonlinear steps can be reconfigured to a \*MANIFEST simulation.
  - Generate dictionaries containing all the nodes and elements in a model, allowing a script to quickly look up their properties.
  - Insert, delete, and replace keyword blocks.
  - Find a keyword block using the keyword name and specific parameters.
  - Find all references to nodes and elements
- Envisioned uses:
  - Powerful keywords editor for **3DEXPERIENCE** input files, or any input file that needs changes not supported by a GUI.
- Intended Users:
  - Field engineers or customers with Python scripting experience.
- Availability:
  - inpRW is distributed through the SIMULIA Community, specifically the [inpRW Main Page](#).

## 1.2 Project Goals

- Parse an input file to make the data contained therein usable to Python, while storing the original formatting.
- Be able to parse every Abaqus keyword type
- **Write out an input file from the inp object that is identical to the original input file, except for any changes the user makes**
  - Optional, can be disabled for better performance.
- Add functions to help the user find data in the input file.
- Add functions to help the user modify data in the input file.
- Provide the user the same flexibility Abaqus provides (case-insensitivity, space-insensitivity) without modifying the data.
- Give users the ability to easily extend the *inpRW* class with their own functions.

## 1.3 Important Limitations

inpRW is meant to parse every Abaqus keyword type. However, it does not yet meet this lofty goal. It has focused on parsing input files **3DEXPERIENCE** can write. If inpRW cannot fully parse a keyword, it will still read and write those keywords accurately, it just does not take the appropriate actions when it encounters some keywords. The following is a partial list of the most notable keyword blocks which are not fully supported:

- **\*ASSEMBLY and the associated keywords (\*PART, \*INSTANCE)**
  - The structure will be recognized, but *nd* and *ed*, set names, etc. won't be accessible via the instance name.
- **\*SYSTEM**
  - Nodal coordinates won't be reported transformed into the active system.
- **\*NGEN, \*ELGEN, \*NFILL, etc.**
  - The entities created by these keywords won't be accessible.

These limitations will most likely be addressed in the future. For now, you can work around them by utilizing flattened input files, which will remove the preceding keywords and generate an equivalent input file. To create and save a flattened input file, you can add the following code to an `abaqus_v6.env` file prior to running a datacheck or full analysis on the input file:

```
print_flattened_file=ON
def onJobCompletion():
    from os import path, system, lstat, listdir
    import osutils
    if path.exists('%s_f.inp' % id):
        src='%s_f.inp'%(id)
        dest=savedir
        osutils.copy(src,dest)
```

The preceding code will create a input file named “JOBNAME\_f.inp”. This will be the flattened input file, which removes the above problematic keywords for inpRW.

For more information on modifying the Abaqus environment files, see section [Using the Abaqus environment files](#) of the Abaqus Installation, Licensing & Configuration guide.



## 1.4 Projects Which Used inpRW

- Convert step data to \*MANIFEST input files
- Convert \*RELEASE to connector element equivalent
- Generate hydro-dynamic load fields
- Integrate with new Keyword Edit feature of 3DEXPERIENCE
- Cut an input file by keeping only the nodes corresponding to a particular nodeset, deleting all features that reference deleted nodes.

## 1.5 Data Structure

When an input file is parsed, it creates an *inpRW* instance. An *inpRW* instance contains many attributes and functions, all of which are fully documented in the *API Reference* section. Here's an outline of how *inpRW* organizes the parsed keyword blocks. This only shows select attributes of the class.

```
inp (inpRW)
├── keywords (inpKeywordSequence)
│   ├── keyword (inpKeyword)
│   │   ├── name (str)
│   │   ├── parameter (csid)
│   │   ├── data (list or Mesh)
│   │   ├── comments (list)
│   │   ├── path (str)
│   │   └── suboptions (inpKeywordSequence)
│   │       ├── keyword (inpKeyword)
│   │       └── ...
│   └── keyword (inpKeyword)
│       └── ...
├── nd (TotalNodeMesh)
├── ed (TotalElementMesh)
├── kwg (csid)
├── steps (csid)
├── jobSuffix (str)
├── inputFolder (str)
├── inpKeywordArgs (dict)
└── outputFolder (str)
```

## 1.6 Project Structure

- *inpRW* is composed of the following sub-modules:
  - *\_inpR*
  - *\_inpW*
  - *\_inpMod*
  - *\_inpFind*
  - *\_inpFindRefs*
  - *\_inpCustom*
  - *\_importedModules*
- There are several other modules used by this class, but are not directly part of the class:
  - *centroid*
  - *config*
  - *csid*
  - *eval2*
  - *inpDecimal*
  - *inpInt*
  - *inpKeyword*
  - *inpKeywordHelper*
  - *inpKeywordSequence*
  - *inpString*
  - *inpRWErrors*
  - *mesh*
  - *misc\_functions*
  - *NoneSort*
  - *printer*
  - *repr2*

## 1.7 Documentation Conventions

- Functions and attributes which start with “\_” are hidden and are mainly used as internal functions of *inpRW*. They are not meant to be called by the end-user. They are documented here for completion and in case they might prove useful for some workflow.
- *Italics* will refer to a *parameter[1]* of a Python function.
- For a term that can have multiple definitions, ‘[#]’ refers to the entry to which the term is referring.
- Abaqus keyword names will be displayed as such: \*STEP. Similarly, Abaqus keyword *parameters[2]* will be in all caps (i.e. NLGEOM).
- The term “inp” will almost always be shorthand to refer to an instance of *inpRW*.

## PROJECT BACKGROUND

I started writing *inpRW* because I frequently needed to edit Abaqus input files, but it was tedious to do so using text editors. Working on the input file itself is convenient because it is well-defined, open, portable, and supports every Abaqus keyword. I wanted a tool that could run on any operating system and that could work on input files generated by any pre-processor.

### 2.1 Difficulties of parsing Abaqus input files

Writing parsers is a difficult task. While simpler than many languages, the Abaqus input file language still has some quirks which make accurate parsing difficult. In particular:

- Abaqus is not case-sensitive nor space-sensitive for keywords and parameters, while Python is sensitive to both.
- Keyword blocks are denoted with one \* character and end with the next keyword block.
- Comments are denoted with two \* characters and can start any line in the input file.
- Keywords and parameters do not need to be fully written out; they only need enough information to uniquely identify them.
- Data formatting varies per keyword, and different parameters can change the data for a given keyword.
- Many keywords directly enhance other keywords, and need to directly follow their parent keywords. There are three different methods for terminating these keyword groups:
  - \*END keyword (\*PART, \*STEP)
  - Keyword that is not a valid suboption of the parent keyword (\*MATERIAL, \*OUTPUT)
  - Read until the end of the file (\*INCLUDE, \*MANIFEST)
- Abaqus has many default values that do not need to be specified.

### 2.2 inpParser Module

The *inpParser* module is included with Abaqus Python. It is essentially a Python API to the input file parser the Abaqus solver uses. Its purpose is to interpret an Abaqus input file in a way the solver can understand, with acceptable performance for massive input files, and execute the job in a manner the user intended. As part of this design, *inpParser* makes changes to the input file data, such as changing the case and spacing of items to provide case- and space-insensitivity, filling in default values of some parameters and data, and ignoring comments that appear after the last keyword and in the data lines of a keyword. There is also no method to write an input file from the parsed data in *inpParser*. Thus, *inpParser* is not a great foundation upon which to build a parsing tool meant to modify input files prior to running Abaqus jobs.

## 2.3 inpRW History

*inpRW* was originally just meant to write out an input file from the data from *inpParser*. However, due to the changes *inpParser* makes to the input data when parsing it, the module could not simply write out a new input file with whatever data was in the *inp* object, as it would not look anything like the original input file. A module to write from an *inpParser* *inp* object would likely require a special case for most keywords and parameter combinations, which would take an inordinate amount of work to create. Therefore, I decided to write a new module for parsing the input file and writing out a new input file, and this of course needed some functions that could find and change data in the input file. *inpRW* was originally written for Python 2.7, so that it could run in Abaqus Python. However, I converted the script to Python 3 for a few reasons.

- Python 2.7 is end of life.
- **3DEXPERIENCE** Keyword Edit tool in its initial form is hard-coded to use a Python 3 interpreter.
- Python 3 will have performance improvements over Python 2.7, and some useful features (such as ordered keys in dictionaries).
- Since the utility merely operates on the text of the input file, it does not need any features provided by the Abaqus Python API for pre-processing Abaqus/CAE models or post-processing Abaqus .odbs.

## INSTALLATION AND USAGE INSTRUCTIONS

There are three options for installing `inpRW`. First, a packaged version with a convenient installer is available. The installer can be used to install `inpRW` to the **3DEXPERIENCE** Python 3.7 interpreter.

Second, if you wish to install `inpRW` to a different Python interpreter, you can use `pip` to install `inpRW` from the `.whl` file.

Third, if you do not wish to install `inpRW` to a Python interpreter (or `venv`), you can simply extract the source files from the `.whl` and set the `PYTHONPATH` variable or modify `sys.path` prior to importing `inpRW`.

### 3.1 Option 1: Using `install_to_3DEXPERIENCE.bat`

`inpRW` can be easily installed to a **3DEXPERIENCE** Python 3.7 interpreter using `install_to_3DEXPERIENCE.bat`. `install_to_3DEXPERIENCE.bat` will check for admin rights and relaunch the command if it does not have them, will open a file-browser so the user can specify the version folder of **3DEXPERIENCE** (the script will then find the 3.7 `python.exe` for that installation), and will finally install `numpy`, `scipy` (requirements of `inpRW`) and `inpRW`.

### 3.2 Option 2: Install `inpRW` from `.whl`

If `inpRW` is to be installed and/or upgraded for a different Python installation, the command is

```
PYTHON -m pip install inpRW*.whl --upgrade
```

`PYTHON` should refer to the desired Python executable, which should be at least version 3.7. This is essentially what `install.bat` does, although that file performs some extra tasks like finding the specific Python interpreter used by the Keyword Edit feature of **3DEXPERIENCE**.

Make sure `pip` is upgraded first, or there might be problems installing `inpRW`. The following command will work:

```
PYTHON -m pip install --upgrade pip
```

### 3.3 Option 3: Extract inpRW from .whl

If you do not want to install inpRW, you can instead extract all the files from inpRW\*.whl to a directory; we'll call it inpRW\_src for this example. You will then need to tell the Python interpreter where to find the utility. You can do this in several ways; here are a few examples for Windows systems:

- Set the PYTHONPATH variable to the inpRW\_src directory. You can set this globally, or add the commands to a .bat file prior to calling the Python interpreter. Example: set PYTHONPATH=PATH\_TO\inpRW\_src
- Modify sys.path to include the path to inpRW\_src. This can be done in a Python script before importing inpRW. Example: `sys.path.insert(0, 'PATH_TO\inpRW_src')`.

### 3.4 Calling inpRW

Have any Abaqus input file available and open a Python interpreter in the directory with the input file. Create an *inpRW* class object with the following commands:

```
import inpRW
inp = inpRW.inpRW('inputFileName.inp')
inp.parse()
```

Please see the *API Reference* for all the available functions and their syntax.

### 3.5 Using inpRW with Keyword Edit in 3DEXPERIENCE

Once inpRW has been installed to 3DEXPERIENCE, you can use inpRW with the Keyword Edit functionality of the Mechanical Scenario app. You'll need to first write a Python script which will edit the input file, and then you can upload that file to 3DEXPERIENCE using the Keyword Edit tool. See the [documentation](#) for more details on Keyword Editing.

Here's a brief video showing how to install and use inpRW with the Keyword Edit feature of 3DEXPERIENCE.

## INPRW TESTS

inpRW has an in-progress test suite. This can be downloaded from [the inpRW Files page](#) of the SIMULIA Community. The test suite utilizes the `unittest` and `doctest` functionality built in to Python. The `doctest` examples which have been completed are written in the documentation strings of the corresponding function. These are actually designed to be run using the `unittest` framework. `test_all.bat` and `test_all.sh` will run all the tests on Windows and Linux, respectively.

*inpRW* has been developed on a Windows 10 machine, using Python 3.7, numpy 1.21.6, and scipy 1.7.3. The following configurations have been able to run the `test_suite` for inpRW:

Table 1: Tested platforms

Operating System	Python Version	Numpy Version	Scipy Version
Windows 10	3.7.9	1.21.5	1.7.3
Windows 10	3.9.10	1.26.0	1.11.3
Windows 10	3.10.0	1.26.0	1.11.3
Windows 10	3.11.6	1.26.0	1.11.3
Windows 10	3.12.0*	1.26.0	1.11.3
Arch Linux, lts kernel 6.1.62-1	3.7.17	1.21.6	1.7.3
Arch Linux, lts kernel 6.1.62-1	3.8.18	1.24.4	1.10.1
Arch Linux, lts kernel 6.1.62-1	3.9.18	1.24.4	1.10.1
Arch Linux, lts kernel 6.1.62-1	3.10.13	1.26.2	1.11.4
Arch Linux, lts kernel 6.1.62-1	3.11.5	1.26.2	1.11.4
Arch Linux, lts kernel 6.1.62-1	3.12.0	1.26.2	1.11.4

*inpRW* is expected to work on any OS and with newer versions of Python, numpy, and scipy, so please report any problems.

\* The inpRW test suite reports SyntaxWarnings for “invalid escape sequences”. These do not appear to be causing a problem, but I have not yet investigated why they are arising.





## RUNNING INPRW TESTS

It is a good idea to run the `inprw` test suite to verify it is configured properly on your system. This can be useful when you first install `inprw`, and also if you make any changes to `inprw` (for example, you should run the test suite to make sure your changes had no unintended consequences). While the test suite is yet incomplete, it still tests a good deal of `inprw`, especially the most basic features like importing the module, parsing some input files, etc.

To run the `inprw` test suite, you should first extract the test suite from `inprw_tests.zip`. Next, you need to modify `test_all.bat` or `test_all.sh` from the extracted test suite. The `INPRW_TEST` variable needs to point to the Python interpreter you would like to use for the tests. This interpreter should be able to import `inprw`. If you are running on Linux, make sure the shell scripts are executable.

Once you've specified the `INPRW_TEST` environment variable in `test_all`, you can simply call `test_all.bat` (or `test_all.sh`) to run all of the existing tests. Of course, you can also see the commands used to the tests in this file, so using one of these files is not required.

The test suite also includes the files `test_all_multi_py.bat` and `test_all_multi_py.sh`. These files are designed to call the test suite on all Python interpreters in a given folder. These are primarily used to test `inprw` prior to shipping the code, but they might be useful to others if they need to test `inprw` against multiple versions of Python. If you wish to use these scripts, you'll need to create a folder of `.bat` or `.sh` files which point to the Python interpreters to test, and modify `test_all_multi_py.*` to point to this folder.

Here's an example `.bat` file which prints the Python, numpy, and scipy versions before calling the Python interpreter with the specified arguments:

```
@echo off
setlocal
set TMPRELPATH=%~dp0\..\py37\Scripts\
pushd %TMPRELPATH%
set TMPABSPATH=%CD%\python.exe
popd
echo.
echo ===== venv info =====
echo Running %TMPABSPATH%
call %TMPABSPATH% -VV
echo numpy
call %TMPABSPATH% -m pip show numpy | grep -E "Version:"
echo scipy
call %TMPABSPATH% -m pip show scipy | grep -E "Version:"
echo =====
echo.

set PYTHONPATH=%~dp0\..\..\src
call %TMPABSPATH% %*
endlocal
```

Here's an example .sh file which accomplishes the same task:

```
#!/bin/sh

PY=/home/erik/inpRW_test/inpRW_tests/venv/py37/bin/python
echo
echo ===== venv info =====
echo Running $PY
$PY -VV
echo numpy
$PY -m pip show numpy | grep -E "Version:"
echo scipy
$PY -m pip show scipy | grep -E "Version:"
echo =====
echo

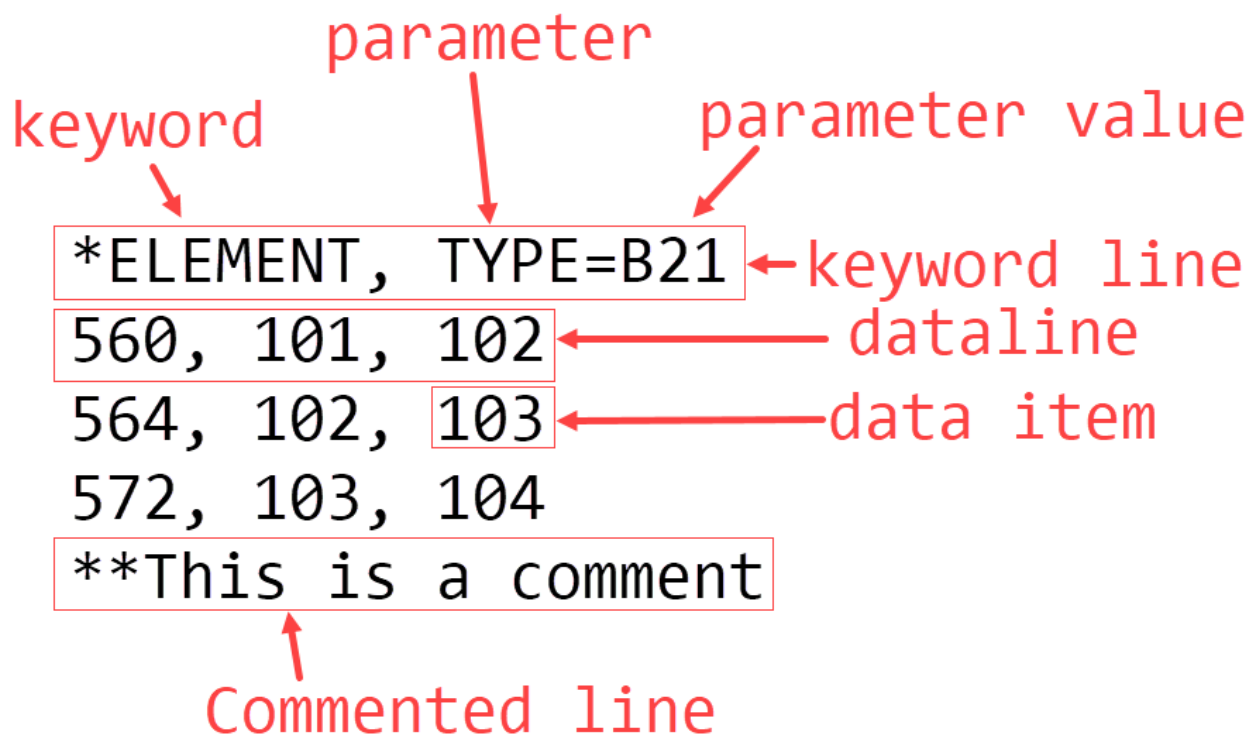
$PY "$@"
```

## INPKEYWORD BASICS

The *inpKeyword* object is the foundation of *inpRW*. It defines the structure of information parsed from the input file, so understanding how the data is stored inside an *inpKeyword* object is necessary to working with *inpRW*.

## 6.1 Abaqus Keyword Block Terminology

First, let's define the different parts of an *Abaqus keyword block*.



A *keyword block* has a *keyword line(s)*, which is composed of a *keyword* and 0 or more *parameters*[2]. Immediately following the *keyword line* are 0 or more *datalines*. The individual *data items* in a *dataline* are separated by commas. Finally, any line in the *input file* can be commented out with '\*\*\*'.

For more information on *Abaqus input files*, please see [Input Syntax Rules](#) for information pertaining to all *keyword blocks*[1], and see the [Abaqus Keywords Guide](#) for information on specific *Abaqus keywords*.

Here is an example of an *Abaqus keyword block*[1]:

```
*ORIENTATION, NAME="Connector1-2Pt1Orientation-1", SYSTEM=RECTANGULAR,
↳DEFINITION=COORDINATES
1., 0., 0., 0., 1., 0.
1, 0.
```

And here is the *inpKeyword* object *inpRW* creates from it (this output has been specially formatted for readability):

```
inpKeyword(name='ORIENTATION',
           parameter=csid(csiKeyString(' NAME'): '"Connector1-2Pt1Orientation-1"',
                                csiKeyString(' SYSTEM'): 'RECTANGULAR',
                                csiKeyString(' DEFINITION'): 'COORDINATES'),
           data=[[1.0, 0.0, 0.0, 0.0, 1.0, 0.0], [1, 0.0]],
           path='self.keywords[10]',
           comments=[],
           suboptions=inpKeywordSequence(num child keywords: 0, num descendant keywords: 0,
↳path: .suboptions))
```

This *inpKeyword* object is accessible via `eval(path)` of the *inpKeyword* object. Note that “self” is only available from commands run from inside *inpRW* itself. If you need to access the item from outside of *inpRW*, you should replace “self” with the instance name. For example, if you call *inpRW* like the following, you would replace “self” with “inp”:

```
import inpRW

inp = inpRW.inpRW('INPUTFILE.inp')
```

## 6.2 Important inpKeyword Attributes

### 6.2.1 Name

*name* corresponds to the *keyword* name.

### 6.2.2 Parameter

*parameter* is a case- and space-insensitive dictionary (*csid*) that contains key-value pairs corresponding to the *parameter[2]* names and their values. If a *parameter[2]* does not have a value assigned to it, the value will be set to ‘’. Since *inpRW* needs to be run in Python 3, the *parameter* dictionary is automatically ordered. Thus, we can easily access the items in the order they were entered. This allows the *parameters[2]* to be written out to a new input file in the original order.

### 6.2.3 Data

*data* is a list of lists that contains the *datalines* for the keyword block. For some keyword blocks, *data* will use a dictionary-like construct. There will be a list which corresponds to each line of *data*. *inpRW* will try to convert each data item to what it deems the appropriate type. If the item appears to be a string, it will leave it as a *string*. If it appears to be a floating point number, it will be converted to a *float* or *Decimal*. If it appears to be an integer, it will be converted to an *integer*.

## 6.2.4 Suboptions

If *inpRW* was parsed with *organize=True*, the *suboptions* field will be populated with the appropriate child keywords, if the keyword block has any valid child keywords. Those subkeywords can in turn have their own subkeywords. This enables *inpRW* to group related keywords together (for example, \*MATERIAL and all the keywords related to that material).

## 6.2.5 Comments

All comment lines associated with this keyword block (i.e. after the keyword line but before the next keyword line) will be stored in this attribute. Each item in *comments* will be of the form [ind, line] where ind is the index of the line in *data* and line is the string of the comment line.

## 6.3 Creating a new inpKeyword object: Recommended option

The most convenient method to create a new keyword object is to first create a string representing the entirety of the keyword block and pass that to the *inputString* parameter of the *inpKeyword* constructor. Also, there will be certain attributes in *inpRW* that control the parsing formatting; you will want to pass these to the new *inpKeyword* instance to ensure the new keyword block is consistent with existing keyword blocks. These attributes are available in *inpKeywordArgs*. The following is an example of the recommended method of creating new *inpKeyword* blocks, assuming an *inpRW* instance “inp” has already been created

```
from inpKeyword import inpKeyword
text = '''*ORIENTATION, NAME="Connector1-2Pt1Orientation-1", SYSTEM=RECTANGULAR,
↳DEFINITION=COORDINATES
1., 0., 0., 0., 1., 0.
1, 0.'''

newKW = inpKeyword(inputString=text, **inp.inpKeywordArgs)
```

This method of creating a keyword block will perform several steps automatically. First, it will parse the entirety of the keyword block according to the settings in *inpKeywordArgs*. Second, it will automatically call several additional functions of *inpKeyword* which determine the type of the keyword block (based on *name*), set the functions which should be used to parse the datalines and format output of the parsed data items, and specify information from the *inpKeyword* block which needs to be written to special attributes of *inpRW* (these attributes will be added when newKW is added to an *inpKeywordSequence* instance).

## 6.4 Creating a new inpKeyword object: Alternate option

When creating a new keyword, perform something like the following:

```
from inpKeyword import inpKeyword
newKW = inpKeyword()
```

This will create a blank *inpKeyword* object, with the following parameters:

```
newKW.name = ''
newKW.parameter = csid({})
newKW.data = []
```

(continues on next page)

(continued from previous page)

```
newKW.path = ''
newKW.suboptions = inpKeywordSequence()
newKW.comments = []
```

Any of the attributes of *inpKeyword* (*inpKeyword.inpKeyword.name*, *inpKeyword.inpKeyword.parameter*, *inpKeyword.inpKeyword.data*, *inpKeyword.inpKeyword.path*, *inpKeyword.inpKeyword.suboptions*, *inpKeyword.inpKeyword.comments*) can be assigned when *newKW* is created, or they can be modified later.

The easiest way to create the *inpKeyword.inpKeyword.parameter csid* is to call *createParamDictFromString()*. Example:

```
inp.createParamDictFromString('NAME="Connector1-2Pt1Orientation-1", SYSTEM=RECTANGULAR,
↳ DEFINITION=COORDINATES')
```

Results in:

```
{'NAME': '"Connector1-2Pt1Orientation-1"',
'SYSTEM': 'RECTANGULAR',
'DEFINITION': 'COORDINATES'}
```

When adding to *data*, make sure it is always a list of lists, even if there is only one line of data. The functions that generate strings from an *inpKeyword* instance assume *data* is a list of lists, and will produce an error or incorrect results if *data* is incorrect. This does not apply to \*NODE or \*ELEMENT keyword blocks, as the data attribute is a *Mesh* instance.

Once you have populated *name*, *parameter*, *data*, *comments*, and possibly *suboptions*, you need to call *\_setMiscInpKeywordAttrs()* and set *\_dataParsed = True*. These will set several important attributes for inserting the keyword into an *inpKeywordSequence* and producing a string from the keyword object.

*path* should not need to be specified manually. Once the new keyword has been placed into *keywords*, you can call *updateInp()*, which includes a call to update the *path* of all keywords.

## 6.5 Creating a new inpKeyword object: Hybrid option

Another technique for creating *inpKeyword* objects is to set *inputString* to a string representing the keyword lines of the new block, and then add the data later. This is possible if you know the keyword name and optionally the parameters before you know the data. You might wish to do this if the data for the keyword block has been calculated based on other information, and thus it is already in the proper format (i.e. you wish to skip converting the data to strings and then parsing those strings). If you instance *inpKeyword* with an *inputStr* specifying the keyword lines, this will automatically set the proper attributes of the class based on the keyword name, so you do not need to manually set them later.

## STRINGS, INTEGERS, AND DECIMALS

One option with `inpRW` is the `preserveSpacing` parameter. If this is `True`, `inpRW` will track every space character, and the exact formatting of every number for every piece of data parsed from the input file.

Since a `float` cannot accurately represent numbers with decimal points ([reference](#)), `inpRW` has a parameter (`useDecimal`) to convert all numbers that would be evaluated to `float` to `Decimal` instead. Setting `useDecimal` to `False` will treat numbers as `float`; this will result in faster performance, but some formatting and value precision will be lost. If `useDecimal = True`, `inpRW` will use `floats` in one location: `pd`.<sup>1</sup>

However, using `Decimal` is not enough by itself to exactly reproduce an input file. Converting a string into its underlying data type will naturally lose spaces and exact formatting of the original number, so `inpRW` uses a custom `eval()` function (`eval2()`), and 3 new classes: `inpDecimal`, `inpInt`, and `inpString`. These functions track the exact spacing and number formatting in the original item and store that information in the `_formatStr` and `_formatExp` attributes of the new items. These classes inherit from `Decimal`, `int`, and `str`, and should support all the same operations as their base classes. Please see those class definitions for more details.

Creating an instance of `inpRW` with `preserveSpacing = False`, `useDecimal = True` will not preserve the exact spacing, and will instead create `Decimal`, `int`, and `str` items. Using `preserveSpacing = False`, `useDecimal = False` will use `float`, `int`, and `str` items. Using `preserveSpacing = True`, `useDecimal = True` will preserve the exact spacing and use `inpDecimal`, `inpInt`, and `inpString`.

Throughout this documentation (and particularly the *API Reference* section), there will be references to `str`, `int`, and `Decimal`. In almost all cases, these classes can be used interchangeably with the `inp*` variant.

---

<sup>1</sup> There is no way to avoid this, as the data in a `*PARAMETER` keyword block is meant to be run as Python code, there is no good way to evaluate the Python commands while turning `floats` into `Decimals`, and even if we did so mathematical operations on those values would differ slightly from the Abaqus job. Calling `_subParam()` will automatically convert any `float` to a `Decimal`, although a `Decimal` of a `float` of a `str` will not be as precise as a `Decimal` of a `str`.





## IMPORTANT INPRW ATTRIBUTES

Since *inpRW* is a large project, here is a summary of the key attributes with which a user might wish to interact.

### 8.1 keywords

*keywords* is the *inpKeywordSequence* that contains the parsed *keyword blocks*. This is the most important attribute of *inpRW*, as all of the information from the input file is stored here.

### 8.2 kwg

Short for keyword group. *kwg* is a *csid* that uses the keyword names in the input file as keys. The value for a given key is a *set* containing all keyword blocks with that name. For example, `inp.kwg['STEP']` will contain all the \*STEP keyword blocks. Since a *set* is naturally unsorted, use *sortKws()* if you need the keywords in their original order.

### 8.3 steps

*steps* is a *csid* with the step names as the key, and the corresponding step keyword block as the value. If the *NAME parameter[2]* is not specified for a keyword block, an integer corresponding to the steps index in `kwg['STEP']` is used as the key.

### 8.4 step\_paths

*step\_paths* is a list that contains the path to each \*STEP keyword block in their proper order.

### 8.5 nd

*nd* is a *TotalMesh* instance, which behaves similarly to a dictionary. It directly references every node in the input file. The keys to *nd* are node labels, and the values are *Node* instances.

## 8.6 ed

*ed* is a *TotalMesh* instance, which behaves similarly to a dictionary. It directly references every element in the input file. The keys to *ed* are element labels, and the values are *Element* instances.

## 8.7 inpName

*inpName* is a string that corresponds to the name of the *input file* (including the extension, but not the full path to the file).

## 8.8 inputFolder

*inputFolder* is a string that represents the path to the *input file*. The value will be parsed from the *inpName parameter[1]* of *inpRW* when instantiating *inpRW* if *inpName* includes the full path. If *inpName* does not contain the path, *inputFolder* will be a blank string. *inputFolder* will be prepended to *inpName* when reading files with *inpRW*.

## 8.9 outputFolder

*outputFolder* is a string that represents the path to which new *input files* will be written. The value will be parsed from the *inpName parameter[1]* of *inpRW* when instantiating *inpRW* if *inpName* includes the full path. If *inpName* does not contain the path, *outputFolder* will be a blank string. *outputFolder* can of course be manually set prior to writing out new input files. *outputFolder* will be prepended to *inpName* when writing new files.

## 8.10 jobSuffix

*jobSuffix* is a string that will be appended to *inpName* (without the file extension) when writing out an *input file* or any reference to a new *input file*. If the *jobSuffix parameter [1]* is not specified when instantiating *inpRW*, it will default to ‘\_NEW.inp’. If using *Keyword Edit* with 3DEXPERIENCE, *jobSuffix* should be set to ‘\_NEW.inp’.

## 8.11 includeFileNames

*includeFileNames* is a list which contains the names of all *input files* read in by the parent *input file*. This will cover \*INCLUDE, \*MANIFEST, and any keyword that reads its data from another *input file*.

## 8.12 includeBlockPaths

*includeBlockPaths* is a list that contains the *path* to each block in *keywords* that reads from a secondary *input file*.

## MISCELLANEOUS USAGE HINTS

*inpRW* is merely a Python utility composed of the main module (*inpRW* and the *\_inp\** modules) and supporting modules and functions. It does nothing by itself. The user will always need to write at least a small script to perform the actual operations. Here is a simple example script that will parse an *input file*, add a *\*CONTROLS* keyword block before the *\*END STEP* keyword block of the first general step, and write out the new *input file*:

```
import inpRW
import inpKeyword
from misc_functions import rsl
from sys import argv

newKwText = '''*CONTROLS, parameter=field
,,,1.0'''

jobName = argv[-1]
inp = inpRW.inpRW(jobName, organize=True, preserveSpacing=True, useDecimal=True,
↳ parseSubFiles=True, jobSuffix='_NEW.inp')
inp.parse()
newKw = inpKeyword.inpKeyword(inputString=newKwText, **inp.inpKeywordArgs)

#this will search for the first general step in the model
stepblock = [step[1] for step in inp.steps.values() if rsl(step[1].suboptions[0].name)
↳ in inp._generalSteps and not 'perturbation' in step[1].suboptions[0].parameter][0]

endstep1path = inp.findKeyword('endstep', parentBlock=stepblock)[0].path #this is a case-
↳ and space-insensitive search, it will always insert the new keyword block before the
↳ *end step keyword block of stepblock
inp.insertKeyword(newKw, path=endstep1path)
inp.writeInp()
```

*inpRW* is intended to contain all the functions necessary to parse the entire *input file* and write out a new *input file* from the parsed structure. Any *input files* that cannot be parsed are either due to a bug and should be reported, or are cases not encountered yet. Initial support prioritizes input files that **3DEXPERIENCE** can produce, so some keywords are not yet supported. For example, keywords like *\*PART*, *\*ASSEMBLY*, and *\*NGEN* are not fully supported yet, and there will likely be odd behavior when trying to access the information stored in these keyword blocks or input files containing them.

*inpRW* also contains several functions for finding data in the *input file*, and some functions for making modifications to the data in the input file. The functions for modifying data were included because they were seen as tasks that many users would want to perform. They will not be sufficient for many projects; users will need to create their own functions to modify the data in the input file.

*inpRW* works mainly on the text of the *input file*. It has almost no other information available to it, except keyword

groupings and element type information. Therefore, it cares very little what is actually in the input file, nor what you write into it. Thus, it is perfectly possible to use this tool to create an input file that will not pass an Abaqus datacheck, as this tool does absolutely no logical checking itself. It is up to the user to use the functions in this utility to produce a working Abaqus input file.

Even though *inpRW* mostly does not care what information is in the *input file*, some pieces of *inpRW* look for specific *keywords* or *parameters*[2]. Those sections of the tool expect the *keyword* or *parameter*[2] to be fully spelled out, which is not a requirement of the Abaqus solver. Thus, *inpRW* will likely fail on input files that do not fully spell out *keywords* or *parameters*[2], although many functions might still work. This limitation might be addressed in future versions of *inpRW*.

The first step to using *inpRW* effectively is to know the content of the *input file*. This means writing out an input file before writing the script. The user should have some idea of the content of the input file to guide development of a script to modify that input file.

Calling *inpRW* with `organize=True` will provide the greatest functionality. Functional *keyword blocks* will automatically be grouped together as *parent blocks* and *suboptions*. Thus, it is possible to search for \*PLASTIC and retrieve the entire \*MATERIAL block with all its sub-blocks, or it is possible to search for \*BOUNDARY in a particular \*STEP. These actions would be much more difficult and computationally expensive if dealing with a flat input file structure. Here are the commands for the two examples described above:

```
matblock = inp.getParentBlock(inp.findKeyword('plastic')[0], parentKW='material')
#inp.findKeyword will return a list of all keyword blocks that match the search criteria;
→ hence the [0] to get just the 0th result.

step = inp.steps['step-1']
boundaryblock = inp.findKeyword('boundary', parentBlock=step)[0]
```

*findKeyword()* is an extremely powerful tool for finding *keywords* in the *keywords*. The user can use this function to search for *keywords* using any or all of *keywordName*, *parameters*, and *data*, and the search can also be performed on only the *suboptions* of a *parent block*. This function will return a list of all *inpKeyword* objects that match the search criteria. The returned *inpKeyword* objects can be used in other functions, as shown in the previous section.

*inpRW* parses information from the *input file* and stores that information (and optionally formatting) in the original form. Thus, spacing and capitalization can be preserved exactly. However, Python itself is case-sensitive and space-sensitive. *inpRW* solves this problem in two ways. First, all data that would be stored in a dictionary should instead be stored in a *csid*. Every dictionary a scripter creates that will be part of an *inpKeyword* should instead be a *csid*. Second, the *rsl()* function takes a string as input and returns an all lowercase string which has had all spaces removed. This function is used throughout *inpRW* whenever a string needs to be compared to another string. Consider the following example:

```
from misc_functions import rsl
a = 'element set 1'
b = '"ELEMENT SET 1"'

if rsl(a) in rsl(b):
    print('yes')
```

In the example above, “a” represents the string for which the user would like to search, and “b” represents the actual value of an element set name in the *input file*. By comparing the *rsl()* results for each item, the user does not need to know the exact formatting of “b” in order to match it. Also, this example does not change the formatting of “b”, so it can be written back to the *input file* with the original formatting.

Users and authors of *inpRW* should use *csid* and *rsl()* extensively, as they are great productivity boosters.

## EXAMPLE SCENARIOS

This guide is intended to show some common and simple operations that can be performed with inpRW. It will introduce you to some of the key functions, but it does not provide the full documentation for these functions. That information can be found in the *API Reference*.

### 10.1 Example input file

This example input file was created in Abaqus/CAE and will be used for all examples mentioned in this article. To follow along, copy the text of the input file and save it as **example\_inp.inp**.

```
*Heading
** Job name: example_inp Model name: Model-1
** Generated by: Abaqus/CAE 2021.HF9
*Preprint, echo=NO, model=NO, history=NO, contact=NO
** -----
**
** PART INSTANCE: Part-1-1
**
*Node
    1,      0.,      0.,      0.
    2,      5.,      0.,      0.
    3,      0.,      5.,      0.
    4,      5.,      5.,      0.
*Element, type=S4R
1, 1, 2, 4, 3
*Nset, nset=Part-1-1_shell_set, generate
1, 4, 1
*Elset, elset=Part-1-1_shell_set
1,
** Section: shell_section
*Shell Section, elset=Part-1-1_shell_set, material=steel
0.5, 5
*System
*Nset, nset=Set-2
2,
*Nset, nset=Set-3
3,
*Nset, nset=Set-4
2, 4
*Nset, nset=origin
```

(continues on next page)

(continued from previous page)

```

1,
**
** MATERIALS
**
*Material, name=steel
*Elastic
210000., 0.3
**
** BOUNDARY CONDITIONS
**
** Name: BC-2 Type: Displacement/Rotation
*Boundary
Set-2, 2, 2
Set-2, 3, 3
Set-2, 4, 4
Set-2, 5, 5
Set-2, 6, 6
** Name: BC-3 Type: Displacement/Rotation
*Boundary
Set-3, 1, 1
Set-3, 3, 3
Set-3, 4, 4
Set-3, 5, 5
Set-3, 6, 6
** Name: fix_origin Type: Displacement/Rotation
*Boundary
origin, 1, 1
origin, 2, 2
origin, 3, 3
origin, 4, 4
origin, 5, 5
origin, 6, 6
** -----
**
** STEP: Step-1
**
*Step, name=Step-1, nlgeom=YES
*Static
1., 1., 1e-05, 1.
**
** BOUNDARY CONDITIONS
**
** Name: Load Type: Displacement/Rotation
*Boundary
Set-4, 1, 1, 0.1
**
** OUTPUT REQUESTS
**
*Restart, write, frequency=0
*Output, field, frequency=1, variable=PRESELECT
**
** HISTORY OUTPUT: H-Output-1

```

(continues on next page)

(continued from previous page)

```

**
*End Step
** -----
**
** STEP: Step-2
**
*Step, name=Step-2, nlgeom=YES
*Static
1., 1., 1e-05, 1.
**
** OUTPUT REQUESTS
**
*Restart, write, frequency=0
*Output, field, frequency=1, variable=PRESELECT
**
** HISTORY OUTPUT: H-Output-1
**
*End Step

```

Then run the following commands to generate the inp object:

```

import inpRW

inp = inpRW.inpRW('example_inp.inp', preserveSpacing=True, useDecimal=True,
↳organize=True)
inp.parse()

```

You will obviously need to know the syntax of any keywords you wish to modify or create, so you will need to use the [Abaqus Keywords Reference Guide](#) to find this information.

This guide will use Python's 0-based indexing for all such numbers.

## 10.2 Modifying data in an existing inpKeyword block

In this example, we will change the initial increment size in Step-1, and set the INC parameter of Step-1. We will use the following steps to accomplish this:

- Find the *inpKeyword* blocks to modify
- Modify the appropriate fields

First, we will change the initial increment size in Step-1. We need to find the *inpKeyword* block for Step-1. In general, we can use *findKeyword()* to find a specific keyword block. However, *inp* has two attributes which are shortcuts for locating steps. *steps* is a case- and space-insensitive dictionary (*csid*) that uses the step name as the key. *kwg* is a *csid* with keys corresponding to the unique keyword names in the input file. For example, *inp.kwg['step']* is a list with all the \*STEP keyword blocks, in the order they appear in the input file. Thus, we can easily select the steps by their name or relative order using one of these two attributes.

For this example, we'll use the *steps* approach. First, let's check the contents of *inp.steps['step-1']*:

```

In : inp.steps['step-1']
Out: [0, inpKeyword(name='Step', parameter=csid({' name': Step-1, ' nlgeom': YES})),
↳data=[], path='self.keywords[16]', comments=[], suboptions=[...] Len 5)]

```

Since the entry contains a list with the relative step index (0) and the *inpKeyword* block, and we want just the last item from the list (the *inpKeyword* block, we can use this command:

```
step1 = inp.steps['step-1'][-1]
```

Please note that *steps* is a *csid* with the step name as the key, and a value of [relative step index, parsed keyword block]. If you would prefer to reference the steps by their relative order, you could use the following command to retrieve the same block:

```
step1 = inp.sortKWs(inp.kwg['step'])[0]
```

Now that we found the **step1** keyword block, we need to find the \*STATIC block in that step. Since we created **inp** with **organize=True**, we can find \*STATIC as the 0th suboption of **step1** since the procedure must immediately follow \*STEP in a valid input file. Then we merely change the 0th item of the 0th data line.

Please recall that floating point like numbers should be specified as *decimals* if *inpRW* has been specified with *useDecimal* = True. We can also use *inpDecimal*, which inherits from *Decimal*, but also captures the exact string formatting of its input string. Since *useDecimal* was set when we instantiated *inpRW*, we will use the following commands:

```
from inpDecimal import inpDecimal
static = step1.suboptions[0]
static.data[0][0] = inpDecimal('0.1')
```

Next, we will add the “INC=1000” parameter to the **step1** *inpKeyword* block. We will accomplish this by creating a new parameter *csid* with just the new parameter, and then updating the existing parameter *csid*. We do not want to replace the parameter field (as was proposed in a previous version of the guide), as this would break any shared memory references to items in the parameter field.

```
newpar = inpKeyword.createParamDictFromString(' INC=1000', ps=True)
step1.parameter.update(newpar)
```

## 10.3 Creating a new keyword block and inserting it into the inp data structure

There are several steps to creating a new *inpKeyword* block and inserting the block into *inp.keywords*.

- Create the new *inpKeyword* and populate the appropriate fields
- Find the desired location in *inp.keywords* to place the new *inpKeyword* block
- Insert the new *inpKeyword*
- Update *inp.keywords*

In this example, we will create a “\*OUTPUT, HISTORY” *inpKeyword* block, and add a “\*ENERGY OUTPUT” suboption. We need to import the *inpKeyword* module to create a new *inpKeyword*. We will use *makeDataList()* to split up the output variables to multiple data lines. We do not need to specify *path* for the *inpKeyword*; this will be handled by *inp.updateInp()*, which will be called when we insert the keyword block.

```
from inpKeyword import inpKeyword
from misc_functions import makeDataList

vars = ['ALLAE', 'ALLCCDW', 'ALLCE', 'ALLCEN', 'ALLCET', 'ALLCCSD', 'ALLCCSDN',
        'ALLCCSDT', 'ALLCD',
```

(continues on next page)



(continued from previous page)

```
'ALLFD', 'ALLIE', 'ALLKE', 'ALLPD', 'ALLSE', 'ALLVD', 'ALLDMD', 'ALLWK', 'ALLKL', 'ALLQB'
↳ ', 'ALLEE',
'ALLJD', 'ALLSD', 'ETOTAL']
eo = inpKeyword.inpKeyword(name='Energy Output', data=makeDataList(vars, 8))
eo._setMiscInpKeywordAttrs()
eo._dataParsed = True
oh = inpKeyword.inpKeyword(name='Output', parameter=inpKeyword.createParamDictFromString(
↳ ' History, Frequency=1'))
oh._setMiscInpKeywordAttrs()
oh._dataParsed = True
oh.suboptions.append(eo)
```

Note that items for the `inpKeyword` can either be specified at creation, or they can be modified afterwards.

Let's also add another keyword block; this one will modify the field controls. We'll use a different technique to create this `inpKeyword` block. We'll specify the entire keyword block as a string, and let the `inpKeyword` instance parse the string and generate the appropriate fields.

```
controlsKwText = '''*CONTROLS, parameter=field
,,,1.0'''
controlsKw = inpKeyword.inpKeyword(inputString=controlsKwText, **inp.inpKeywordArgs)
```

We pass `inp.inpKeywordArgs` to the `inpKeyword` instance to ensure we use the same parsing settings we used for the rest of the input file.

Next, we need to find the right place to insert the new keyword blocks. Let's insert it before `*END STEP` in `step1`.

```
step1 = inp.steps['Step-1'][-1] #step1 was assigned earlier in this tutorial, but it's
↳ included here for reference
steplend = inp.findKeyword(keywordName='End Step', parentBlock=step1)[0]
path = steplend.path
inp.insertKeyword(controlsKw, path=path, updateInpStructure=False)
inp.insertKeyword(oh, path=path)
```

Note that `findKeyword()` returns a list, so we grab the 0th (and only item if we setup the search properly). Setting `parentBlock` for `findKeyword()` forces the function to check only suboptions of `parentBlock`.

Also note that `insertKeyword()` will place the new keyword ahead of whatever was at `path`. Thus, `controlsKw` will be in front of `steplend`, and then `oh` will be in front of `controlsKw`.

Finally, note that `insertKeyword()` command will by default update `path` of every `inpKeyword` (and their `suboptions`) in `keywords` after the location of the inserted keyword. This can be turned off (set the `updateInpStructure` parameter to False) if you need to insert a lot of keywords and want to update after inserting all keywords.

## 10.4 Deleting a keyword block

Deleting a keyword block is a relatively simple process.

- Find the keyword block to delete
- Call `deleteKeyword()`

For this example, we will delete the `*RESTART` keyword blocks, as they are not used in this analysis, and the `*OUTPUT, FIELD` request in the second step, as it is unnecessary.

First, we find the `*RESTART` keyword blocks.

```
rblocks = inp.findKeyword(keywordName='Restart')
rblocks.reverse()
```

`findKeyword()` will return the valid blocks in the order it found them in the input file. Recall that `keywords` inherits from the list type. If we remove an item from a list, all items after that item will be moved to a different index. Thus, we reverse `rblocks` so we can delete from the back of the sequence, as subsequent blocks to be deleted will be ahead of the previously deleted block in `keywords`.

Please note that you will normally not be able to sort a list of keywords using the default functions assorted with `list` types if the input file was parsed with `Organize = True`. The keywords need to be sorted by their `path` attributes, and this will be too complicated for the built-in ordering functions. `sortKWs()` will sort any sequence of keywords in their proper order. `findKeyword()` calls `sortKWs()`, so the keyword list will already be in order.

We will delete the keywords by passing `path` from a keyword block to the `path` parameter of `deleteKeyword()`.

```
for block in rblocks:
    inp.deleteKeyword(path=block.path, updateInpStructure=False)
inp.updateInp()
```

`deleteKeyword()` normally calls `updateInp()` when it executes, but we turned it off in the loop. We manually call it after we deleted all the desired keyword blocks. Updating the inp structure can take a significant amount of time for a large input file, especially if it is called multiple times unnecessarily.

Finally, we repeat the same process, except we are deleting just one block, the \*OUTPUT, FIELD keyword block in the last step. Here are the commands for this task:

```
ofblock = inp.findKeyword(keywordName='Output', parentBlock=inp.sortKWs(inp.kwg['step
↪ '])[-1])[0]
inp.deleteKeyword(path=ofblock.path)
```

## 10.5 Writing the new input file

Now that we've made changes to the input file data structure, we merely need to write it out using the following command:

```
inp.writeInp()
```

This will create a new input file using the original name, but ending with “\_NEW.inp”. If the original job was multiple files (i.e. \*INCLUDE), `writeInp()` will write the data out to the same relative file structure, while appending `jobSuffix` to each file and adjusting the \*INCLUDE keyword blocks in the parent file.

## 10.6 Complete Python Script

Here are all of the python commands used in this guide in the form of a single script:

```
import inpRW
import inpKeyword
from inpDecimal import inpDecimal
from misc_functions import makeDataList

inp = inpRW.inpRW('example_inp.inp', preserveSpacing=True, useDecimal=True, ↵
```

(continues on next page)

(continued from previous page)

```

↳organize=True)
inp.parse()
step1 = inp.steps['step-1'][-1]
static = step1.suboptions[0]
static.data[0][0] = inpDecimal('0.1')
newpar = inpKeyword.createParamDictFromString(' INC=1000', ps=True)
step1.parameter.update(newpar)
vars = ['ALLAE', 'ALLCCDW', 'ALLCCE', 'ALLCCEN', 'ALLCCET', 'ALLCCSD', 'ALLCCSDN',
↳ 'ALLCCSDT', 'ALLCD',
'ALLFD', 'ALLIE', 'ALLKE', 'ALLPD', 'ALLSE', 'ALLVD', 'ALLDMD', 'ALLWK', 'ALLKL', 'ALLQB
↳ ', 'ALLEE',
'ALLJD', 'ALLSD', 'ETOTAL']
eo = inpKeyword.inpKeyword(name='Energy Output', data=makeDataList(vars, 8))
eo._setMiscInpKeywordAttrs()
eo._dataParsed = True
oh = inpKeyword.inpKeyword(name='Output', parameter=inpKeyword.createParamDictFromString(
↳ ' History, Frequency=1'))
oh._setMiscInpKeywordAttrs()
oh._dataParsed = True
oh.suboptions.append(eo)
controlsKwText = '''*CONTROLS, parameter=field
,,1.0'''
controlsKw = inpKeyword.inpKeyword(inputString=controlsKwText, **inp.inpKeywordArgs)
steplend = inp.findKeyword(keywordName='End Step', parentBlock=step1)[0]
path = steplend.path
inp.insertKeyword(controlsKw, path=path)
inp.insertKeyword(oh, path=path)
rblocks = inp.findKeyword(keywordName='Restart')
rblocks.reverse()
for block in rblocks:
    inp.deleteKeyword(path=block.path, updateInpStructure=False)
inp.updateInp()
ofblock = inp.findKeyword(keywordName='Output', parentBlock=inp.sortKWs(inp.kwg['step
↳ '])[-1])[0]
inp.deleteKeyword(path=ofblock.path)
inp.writeInp()

```

## 10.7 Modified Input File

Here is the new input file, with all the changes performed in this guide:

```

*Heading
** Job name: example_inp Model name: Model-1
** Generated by: Abaqus/CAE 2021.HF9
*Preprint, echo=NO, model=NO, history=NO, contact=NO
** -----
**
** PART INSTANCE: Part-1-1
**
*Node

```

(continues on next page)

(continued from previous page)

```

      1,          0.,          0.,          0.
      2,          5.,          0.,          0.
      3,          0.,          5.,          0.
      4,          5.,          5.,          0.
*Element, type=S4R
1, 1, 2, 4, 3
*Nset, nset=Part-1-1_shell_set, generate
  1, 4, 1
*Elset, elset=Part-1-1_shell_set
  1,
** Section: shell_section
*Shell Section, elset=Part-1-1_shell_set, material=steel
0.5, 5
*System
*Nset, nset=Set-2
  2,
*Nset, nset=Set-3
  3,
*Nset, nset=Set-4
  2, 4
*Nset, nset=origin
  1,
**
** MATERIALS
**
*Material, name=steel
*Elastic
210000., 0.3
**
** BOUNDARY CONDITIONS
**
** Name: BC-2 Type: Displacement/Rotation
*Boundary
Set-2, 2, 2
Set-2, 3, 3
Set-2, 4, 4
Set-2, 5, 5
Set-2, 6, 6
** Name: BC-3 Type: Displacement/Rotation
*Boundary
Set-3, 1, 1
Set-3, 3, 3
Set-3, 4, 4
Set-3, 5, 5
Set-3, 6, 6
** Name: fix_origin Type: Displacement/Rotation
*Boundary
origin, 1, 1
origin, 2, 2
origin, 3, 3
origin, 4, 4
origin, 5, 5

```

(continues on next page)

(continued from previous page)

```

origin, 6, 6
** -----
**
** STEP: Step-1
**
*Step, name=Step-1, nlgeom=YES, INC=1000
*Static
0.1, 1., 1e-05, 1.
**
** BOUNDARY CONDITIONS
**
** Name: Load Type: Displacement/Rotation
*Boundary
Set-4, 1, 1, 0.1
**
** OUTPUT REQUESTS
**
*Output, field, frequency=1, variable=PRESELECT
**
** HISTORY OUTPUT: H-Output-1
**
*Output, History, Frequency=1
*Energy Output
ALLAE,ALLCCDW,ALLCCE,ALLCCEN,ALLCCET,ALLCCSD,ALLCCSDN,ALLCCSDT
ALLCD,ALLFD,ALLIE,ALLKE,ALLPD,ALLSE,ALLVD,ALLDMD
ALLWK,ALLKL,ALLQB,ALLEE,ALLJD,ALLSD,ETOTAL
*CONTROLS, parameter=field
,,1.0
*End Step
** -----
**
** STEP: Step-2
**
*Step, name=Step-2, nlgeom=YES
*Static
1., 1., 1e-05, 1.
**
** OUTPUT REQUESTS
**
*End Step

```



## GLOSSARY

### 3DExperience

Dassault Systèmes' 3DExperience Platform has apps from every Dassault Systèmes brand. These apps (which focus on different domains) all access a common database. For example, CAD (CATIA) and simulation (SIMULIA) apps access and enhance the same product, which enables collaboration and lifecycle management.

For the purposes of *inpRW*, 3DExperience provides a *pre-processor* that can generate an *Abaqus input file*.

### Abaqus

Dassault Systèmes' primary Finite Element Analysis product suite. It's primary use is nonlinear structural simulations. Abaqus is composed of several products, as described [here](#). As *inpRW* is focused on the Abaqus *input file*, this documentation will primarily use *Abaqus* to refer to the Abaqus solvers.

### data

Can refer to the *data* attribute of *inpKeyword*, or it can just mean “information” in general. If the term is not linked to *data*, it is most likely used in a general term and does not have a specific meaning.

### data item

One item from a *dataline*. See *Abaqus Keyword Block Terminology* for a diagram.

### dataline

One line from the data of a *keyword block[1]*. See *Abaqus Keyword Block Terminology* for a diagram.

### inpParser

The module for parsing Abaqus input files that is included with Abaqus Python. Details on limitations of inpParser here: *inpParser Module*. *inpParser Documentation*.

### input file

A text file with the “.inp” extension. The input file defines the *Abaqus* simulation. This includes information like nodes, elements, sets, surfaces, contact definitions, boundary conditions, loading, and many other options.

### job

An *Abaqus* simulation. The details of the *job* are determined by the *input file*.

### keyword

A keyword is the name that starts an *Abaqus keyword block*. It is preceded by “\*” and is separated from the *parameters[2]* by a comma. A keyword can be more than one word. Please see the *Abaqus Keywords Guide* for the list of valid *Abaqus* keywords.

### keyword block block

1. A section of an *Abaqus input file* consisting of a *keyword line* and 0 or more *datalines*.
2. An *inpKeyword* instance that was parsed from a *keyword block[1]* of an *Abaqus input file*.

### Keyword Edit

Keyword Edit is a function of 3DExperience simulation scenario apps (like Mechanical Scenario Creation) that

allows users to make changes to the *Abaqus input file* that *3DExperience* writes before the *job* is submitted. This can use an .xml file, or a Python script.

[Keyword Edit Documentation](#).

**keyword line**

A line in the input file starting with a *keyword* and also containing *parameters[2]*. A *keyword line* can be continued on subsequent lines if the *keyword* and *parameters[2]* will not fit on one line. If a *keyword line* ends with a comma, *Abaqus* will continue reading *parameters[2]* from the next line.

See *Abaqus Keyword Block Terminology* for a diagram.

**parameter**

1. Input to a Python function.
2. An Abaqus keyword parameter.
3. The *parameter* attribute of an *inpKeyword* object, which holds the parsed Abaqus parameters for a keyword block.
4. An Abaqus keyword. See *\*PARAMETER*.

**parent block**

A *keyword block* that hosts one or more *subblock* in its *suboptions* attribute. *\*MATERIAL* is an example of an *Abaqus keyword block* that can be a *parent block*.

**pip**

pip is the package installer for Python. [pip documentation](#).

**positionl**

positionl stands for position list. It is a list of integers and sequences that represent the path to the object. The format is [keyword, [suboption0, suboption1, ...], [dataline, datapos]].

**pre-processor**

A GUI tool for generating an *Abaqus input file*. Two examples are Abaqus/CAE and the SIMULIA apps in *3DExperience*. Pre-processors are necessary to create meshes, surfaces, and sets on complex geometry, and should be designed to write valid *input files*. There is no pre-processor that can write every keyword the *Abaqus* solvers support, though, so editing the *input file* through means such as *inpRW* is sometimes necessary or convenient.

**suboption****subblock****subkeyword**

A *keyword block* that is a child to a *parent block*. A *\*PLASTIC keyword block* is a valid *subblock* for *\*MATERIAL*.

**user script**

A Python script written by an end-user which imports functionality from *inpRW* to accomplish a certain task. As *inpRW* is merely a collection of useful classes, functions, etc., a user script will always be needed to utilize the built-in functionality of *inpRW* and complete the tasks required for the user's workflow.



## 12.1 Package Contents

### 12.1.1 Classes

```
class inpRW.inpRW(inpName, organize=True, ss=False, rmtrailing0=False, jobSuffix='_NEW.inp',  
                  parseSubFiles=True, preserveSpacing=True, useDecimal=True, _parentINP="",  
                  _parentblock="", _debug=False)
```

Bases: [Read](#), [Write](#), [Mod](#), [Find](#), [FindRefs](#), [Custom](#)

inpRW is a collection of Python modules for parsing, modifying, and writing Abaqus input files.

The inpRW class is composed of [\\_inpR](#), [\\_inpW](#), [\\_inpMod](#), [\\_inpFind](#), [\\_inpFindRefs](#), [\\_inpCustom](#), and [\\_importedModules](#). [\\_inpR](#) has the functions for parsing data from the .inp. [\\_inpW](#) has functions for writing the data in the [inpKeyword](#) object to an input file. [\\_inpMod](#) has functions for modifying the data in the inp object structure. [\\_inpFind](#) has functions for finding information in [keywords](#). [\\_inpFindRefs](#) has functions to search for particular named items and all their references in the input file (i.e. node 1). [\\_inpCustom](#) is empty; whatever functions the end user adds to this module will extend the class. [\\_importedModules](#) contains the module imports for all the [\\_inp\\*](#) modules.

```
__init__(inpName, organize=True, ss=False, rmtrailing0=False, jobSuffix='_NEW.inp',  
         parseSubFiles=True, preserveSpacing=True, useDecimal=True, _parentINP="",  
         _parentblock="", debug=False)
```

[\\_\\_init\\_\\_\(\)](#) is called when [inpRW](#) is instantiated and it sets many instance variables before the input file is parsed.

#### Parameters

- **inpName** (*str*) – The file name and optionally path to the input file.
- **organize** (*bool*) – Will group keywords together as parentblock and suboptions if True. Defaults to True.
- **ss** (*bool*) – Strip spaces from items in input file if True. Defaults to False.
- **rmtrailing0** (*bool*) – Remove trailing 0s from decimal numbers if True. Defaults to False.
- **jobSuffix** (*str*) – Suffix to be appended to each input file (and reference to input files) that inpRW writes out. Defaults to '\_NEW.inp'.
- **parseSubFiles** (*bool*) – Parse sub input files (i.e. target of INCLUDE) if True. Defaults to False.
- **preserveSpacing** (*bool*) – Preserve the original spacing for every item in the input file if True. Defaults to True.
- **useDecimal** (*bool*) – If True, [Decimal](#) (or [inpDecimal](#) if [preserveSpacing](#) = True) will be used instead of [float](#) for all floating point numbers. Will be True if [preserveSpacing](#) is True. Defaults to True.

- **\_parentInp** (*inpRW*) – An *inpRW* instance to serve as the parent for the new instance. Meant to be used when *inpRW* instances itself recursively to handle child input files (i.e. \*INCLUDE).
- **\_parentblock** (*inpKeyword*) – An *inpKeyword* block which will serve as the parent to the new *inpRW* instance. All keywords in the child *inp* will be suboptions to *\_parentblock*.
- **\_debug** (*bool*) – If True, prints some additional information while *inpRW* is performing certain tasks and keeps *\_kw* after parsing the input file. Defaults to False.

**Returns***inpRW* instance**parse()**

This function parses the input file.

**printDocs()**

This function will print the documentation strings for all functions inside the *inpRW* class.

**updateInp**(*gkw=True, b=None, startIndex=0, parentBlock='', updatePaths=True*)

This function calls *updateObjectsPath()*, *\_getLastBlockPath()*, *findStepLocations()*, *\_groupKeywordBlocks()*, *\_findIncludeFileNames()*, and updates *\_\_methods\_\_*, *\_\_members\_\_*, *\_couplingSurfNames*, *\_kinCoupNsetNames*, and *\_distCoupElsetNames*.

**Parameters**

- **gkw** (*bool*) – If True, calls *\_groupKeywordBlocks()*. Defaults to True.
- **b** (*list*) – Passed to *\_groupKeywordBlocks()*. See that function for details. Defaults to None.
- **startIndex** (*int*) – Passed to *updateObjectsPath()*. See that function for details. Defaults to 0.
- **parentBlock** (*inpKeyword*) – Passed to *updateObjectsPath()*. See that function for details. Defaults to ‘’.
- **updatePaths** (*bool*) – Updates *path* in all keyword blocks after *startIndex* if True. Defaults to True.

**\_\_repr\_\_()**

Produces a representation of the *inpRW* instance.

This representation can be called to produce another *inpRW* instance. This will only include the arguments to *inpRW* which contain non-default values. This string does not guarantee producing an identical *inpRW* instance upon calling *parse()*, as the applicable attributes might have changed after *parse()* was called, and there are additional attributes which affect parsing which are not output by this function.

**Returns**

The form will be *inpRW(inpName='path/name', arg=value)*. Additional args will be included only if their value differs from the default value.

**Return type***str***\_\_str\_\_()**

Produces a string of the *inpRW* instance.

This string is meant to help identify the *inpRW* instance. It will always include *inpName* and *jobSuffix*, and will include *inputFolder*, *outputFolder*, *\_parentINP*, and *\_parentblock* only if they are non-empty.

**Returns***str*

## Attributes

**delayParsingDataKws:** `set`

A set for the user to specify the element names which will not have their keywords parsed. The keyword names should be all lower case and include no spaces.

**ed:** `TotalElementMesh`

Element dictionary. Identical in concept to `nd`, but holds element information. See `nd` for further details.

**fastElementParsing:** `bool = True`

Parse element blocks with the fast mode if True and if the number of nodes to define the element type is less than 10.

**includeBlockPaths:** `list = []`

Contains the paths to all keyword blocks which reference child input files. This will be populated as the input file is parsed.

**includeFileNames:** `list = []`

Contains the names of all input files referenced by this input file. This will be populated as the input file is parsed.

**inpKeywordArgs:** `dict`

A dictionary containing the attributes from `inpRW` that should be passed to the `inpKeyword` constructor.

**inpName:** `str`

Name of the input file without extension.

**inputFolder:** `str`

Name of the folder containing the input file, populated automatically if `inpName` passed to `inpRW` includes a path.

**jobSuffix:** `str = '_NEW'`

Suffix to append to each new input file. This should include the file extension.

**keywords:** `inpKeywordSequence`

Holds all the parsed keyword blocks. This is the main point of interaction for the user.

**ktridd:** `set = set()`

A `set` containing the Keywords To Remove If Data Deleted from them. Meant to be used with `deleteItemReferences()`. Can be updated with `updateKTRIDD()`.

**kwg:** `csid`

A `csid` that groups keyword blocks by their keyword name.

**namedRefs:** `csid`

A `csid` to track references to named objects. The keys will be the names of keyword blocks which can create named references. For example, 'NSET'. Each value will be the block which defines the named reference. This `csid` will be populated during `parse()`.

**nd:** `TotalNodeMesh`

Node dictionary. A dictionary type object which contains every node in the input file. `nd` is populated from `Mesh`, which make up the `data` attribute of \*NODE keyword blocks. When operating on nodes, the user should access them through `nd`. When adding new nodes, they should be added to an existing `Mesh`, or create a new \*NODE keyword block with a new `data`, and then update `nd` with the `data` of the new block.

**nodesUpdatedConnectivity:** `list`

Tracks nodes which have had their connectivity updated (i.e. they are connected to fewer elements than they previously were, likely due to the user running `deleteItemReferences()`).

**organize:** `bool = True`

If True, keyword blocks will be grouped together as parent blocks and suboptions.

**outputFolder:** `str`

Name of the folder to which new input files will be written, populated automatically if `inpName` passed to `inpRW` includes a path.

**parseSubFiles:** `bool = True`

Parse sub input files if True.

**pd:** `csid = csid()`

Parameter dictionary, uses each variable defined in \*PARAMETER block datalines as the key, the value is the evaluated result. If the user needs the value of a group of items which may reference parameters, call `_subParam()` on the group of objects.

**pktrid:** `csid = csid()`

A `csid` containing the Parent Keywords To Remove If Child Deleted. Meant to be used with `deleteItemReferences()`. Can be set with `updatePKTRICD()`.

**preserveSpacing:** `bool = True`

Preserve the exact spacing in the input file if true. If that is not important, set to False for a performance improvement.

**rmtrailing0:** `bool = False`

Remove the trailing 0s (i.e. 1.0 -> 1.) to reduce file-size.

**ss:** `bool = False`

Strip spaces from strings.

**step\_paths:** `list`

Stores the `path` to every \*STEP keyword block, in order of appearance in the input file.

**steps:** `csid`

Provides quick access to the steps in `keywords` if the step names are known. The key is the step name and the value is `[index in inp.sortKWs(kwg['step']) , inpKeyword]`. If a step does not have a name, the index is used for the key.

**useDecimal:** `bool = True`

Use `Decimal` for all floating point numbers if True. Will be True if `preserveSpacing` is True.

#### Private Attributes

**\_ParamInInp:** `bool = False`

If True, indicates that \*PARAMETER is a keyword block in the input file. Will be set in `parse()`.

**\_\_members\_\_:** `list`

Holds all the member names of the `inpRW` instance.

**\_\_methods\_\_:** `list`

Holds all the method names of the *inpRW* instance.

**\_addSpace:** `str`

Adds a space between items when *ss* is True.

**\_bad\_kwblock:** `list = []`

For development purposes.

**\_couplingSurfNames:** `set`

A set to track the surface names which are used by \*COUPLING keywords.

**\_curBaseStep:** *inpKeyword* = `None`

Tracks the current base step in the model.

**\_debug:** `bool`

Stores debug mode status.

**\_delayedPlaceBlocks:** `dict = {0: [], 1: [], 2: []}`

Tracks blocks that should be parsed on later passes. Any blocks placed into *\_delayedPlaceBlocks* will have the data in their *\_inpItemsToUpdate* inserted at the end of the *parse()* function. For example, \*PARAMETER keyword blocks should have their data placed before all other blocks (loop 0), and all \*NODE blocks should have their data populated before any \*ELEMENT blocks are placed. Unless otherwise specified in *\_keywordsDelayPlacingData*, all blocks will have their data placed in loop 1.

**\_distCoupElsetNames:** `set`

A set to track the elset names which are used by \*DISTRIBUTING COUPLING keywords.

**\_firstItem:** `bool = True`

Tracks the first item to be written to an output input file; a new line character will not be written prior to the item if True.

**\_inpText:** `str`

Holds the raw text string of the entire input file. Will be deleted once *\_kw* is generated unless *\_debug* = True

**\_joinPS:** `str`

The string which is used to join items together when writing strings from the parsed data. If *ss* is True, this will be ' ', else it is '' (spaces are assumed to be tracked with each item in this case).

**\_kinCoupNsetNames:** `set`

A set to track the nset names which are used by \*KINEMATIC COUPLING keywords.

**\_kw:** `list`

A list of strings, with each string corresponding to a keyword block. If *\_debug* = False, *\_kw* will be deleted once all blocks have been parsed.

**\_kwsunFile**

File object to which a summary of the parsed keyword blocks will be written. Reference *file object*.

**\_kwsunName:** `str = inpName`

Name of the file to which a summary of the parsed keyword blocks will be written.

**\_lbp:** `str`

The path to the last block in *keywords*.

**\_leadingcomments:** `str = None`

Stores any comments prior to the first keyword block.

**\_manBaseStep:** `inpKeyword = None`

The last base step for \*MANIFEST simulations.

**\_nl:** `str`

Tracks the new line character used by the file. If new line characters are inconsistent, *\_nl* will be set to '\n'.

**\_numCpus:** `int = 1`

Indicates the number of cpus to use for multi-threaded supported tasks, which is currently parsing keyword blocks. It is not recommended to use more than 1 cpu, as the current speedup does not justify the extra hardware use.

**\_parentINP:** `inpRW`

References the parent inp instance if this is reading a sub input file (i.e. \*INCLUDE, \*MANIFEST). This is needed to track some variables from the parent instance.

**\_parentblock:** `inpKeyword`

References the parent block. All keyword blocks read by a child inpRW instance will be suboptions of *\_parentblock*.

**\_parentkws:** `list = []`

Tracks the open parent keyword blocks.

**\_subP:** `bool = False`

When evaluating particular items, check if item is a parameter and substitute the true value. If the input file has \*PARAMETER keyword block, this will be set to True automatically.

**\_subkwin:** `int = 0`

Tracks the active suboptions index.

**\_tree:** `scipy.spatial.cKDTree = None`

A KDTree of some portion of the nodes in the input file. Created via *findCloseNodes()*.

## 12.2 Submodules

All of the functions in the submodules will be accessible directly through the *inpRW* instance. For example, *createPathFromSequence()* can be accessed via *inp.createPathFromSequence* instead of *inp.\_inpR.Read.createPathFromSequence*. The submodules are merely for organizational purposes.

## 12.2.1 inpRW.\_inpR

### Module Contents

This module contains functions for parsing the data in the input file.

#### **class** inpRW.\_inpR.Read

The *Read* class contains functions related to reading information from the input file.

#### **createPathFromSequence**(*seq*, *base*='self')

Generates a path string from a sequence of sequences.

This function takes a sequence of sequences representing the keyword position, suboptions (optional), and data (optional). It will generate a path string from the sequence. If calling this function from inside the *inpRW* class, do not specify *base*. If calling it outside the class, *base* can be set to the instance name of *inpRW*.

#### Example

```
>>> import inpRW
>>> inp = inpRW.inpRW('dummy.inp')
>>> inp.createPathFromSequence([0])
'self.keywords[0]'
```

If we need to specify a series of suboptions locations, we use a sublist as the first entry of *seq*. We can also specify a different base string:

```
>>> inp.createPathFromSequence([0, [0, 1]], base='inp')
'inp.keywords[0].suboptions[0].suboptions[1]'
```

Finally, we can specify the path to data items in the 2nd entry of *seq*. Make sure to include a blank list for the suboptions entry if you do not need to specify a sub-keyword block. Negative integers are allowed:

```
>>> inp.createPathFromSequence([-1, [], [0, 1]])
'self.keywords[-1].data[0][1]'
```

#### Parameters

- **seq** (*list*, *tuple*) – A list or tuple of the form [0, [0,1], [0,1]] or [0, [0,1], 0]. Item 0 represents the keyword index, item 1 is optional and represents the suboption indices, item 2 is optional and represents the data position(s). Defaults to [None, [], None].
- **base** (*str*) – A string that will be the base of the output path. Defaults to 'self'.

#### Returns

A string containing the path to the object as specified by *seq*.

#### Return type

*str*

#### **parsePath**(*path*)

Returns a list containing the object at *path* and a list of integers representing the path. The object should be somewhere in the keywords -> subkeywords -> data structure (i.e. it should represent a keyword block, or a data item in a keyword block).

Here's an example. First, we need to parse the input file:

```
>>> import inpRW
>>> inp = inpRW.inpRW('example_inp.inp')
>>> inp.parse()
```

Now, we can call the function and print the keyword block and the position list (positionl):

```
>>> block, positionl = inp.parsePath('self.keywords[16].suboptions[2]')
>>> print(block)

*Restart, write, frequency=0
>>> positionl
[16, [2], None]
```

We can also retrieve a particular data item:

```
>>> inp.parsePath('self.keywords[12].suboptions[0].data[0][0]')
(inpDecimal('210000.'), [12, [0], [0, 0]])
```

If there is no object at the specified path, this will return (None, indices):

```
>>> inp.parsePath('self.keywords[18]')
No keyword block at self.keywords[18].
(None, [18, [], None])
```

#### Parameters

**path** (*str*) – Should be the *path* string from an *inpKeyword* object.

#### Returns

A list containing the object at path and a list of lists containing integers representing the keyword index and possibly suboptions and data indices.

#### Return type

*list*

#### sortKWs(*iterable*, *reverse=False*)

This function will sort the keyword blocks in *iterable* by their top-down order in the input file.

It is needed to sort keywords properly when using *organize = True* when instantiating *inpRW*.

#### Parameters

- **iterable** (*list*) – A sequence of *inpKeyword* blocks.
- **reverse** (*bool*) – If True, sort the keyword blocks in reverse order. Defaults to False.

#### Returns

The keyword blocks sorted in the top-down (or reversed) order as they appear in the input file.

#### Return type

*list*

#### subBlockLoop(*block*, *func*)

Performs *func* on all items in *block.suboptions*.

This will perform *func* on all subblocks in *block.suboptions*. *func* should be a function reference.

Example:



```

self._potentialBlocks = []
pBa = self._potentialBlocks.append
for block in self.keywords:
    pBa(block)
    if block.suboptions:
        self.subBlockLoop(block, pBa)

```

#### Parameters

- **block** (*inpKeyword*) – An *inpKeyword* object with *suboptions*
- **func** (*function*) – A function which will be run on all blocks in *block.suboptions*. Thus, *func* must accept a single *inpKeyword* as an input.

**updateObjectsPath**(*startIndex=0*, *\_parentBlock=""*)

Updates the *path* string of each keyword block after *startIndex*.

This should be called after inserting, deleting, or replacing keyword blocks. It is called in *insertKeyword()*, *deleteKeyword()*, and *replaceKeyword()*, unless it is toggled off.

*\_parentBlock* is intended to be used when this function calls itself recursively, so that nested suboptions will also have their paths updated.

#### Parameters

- **startIndex** (*int*) – The index of a keyword block in *keywords*
- **\_parentBlock** (*inpKeyword*) – When *\_parentBlock* is specified, this function will loop through *\_parentBlock.suboptions* and update those paths.

This function is called by *updateInp()*.

**\_createSubKW**(*block*)

Checks if the keyword name in *block* matches a key in *\_subBlockKWs*. If so, a subkw block, and subsequent keyword blocks will be placed into *suboptions* of this keyword block if they are valid suboptions.

#### Parameters

- **block** (*inpKeyword*) – An *inpKeyword* object.

**\_endSubKW**(*block*)

This function tries to close any and all open sub keyword blocks.

It compares the values of *\_subBlockKWs* with the name of *block* to determine if *block* can close the open parent block.

#### Parameters

- **block** (*inpKeyword*) – An *inpKeyword* object.

**\_getLeadingCommentsPattern**()

This function finds the location where the first keyword block begins in an input file and returns a regular expression pattern to identify that location.

The pattern will be used by *\_splitKeywords()* to split the input file immediately before the start of the first keyword block.

#### Returns

A string representing the re pattern which will be used to split the text of the input file on the characters prior to the first keyword block.

**Return type**

str

**\_groupKeywordBlocks**(*b=None, parentBlock=""*)

Creates *kwg*, with the unique keyword names in the input file as the keys, and a set of blocks as each value.

*kwg* can be used to quickly find all keyword blocks of a given keyword name.

**Parameters**

- **b** (*list*) – A list of *inpKeyword* *blocks*. If specified, only the blocks in *b* will be updated in *kwg*.
- **\_parentBlock** (*inpKeyword*) – Only meant to be specified when this function calls itself recursively, so subblocks are included in *kwg*.

This function is called by *updateInp()*.

**\_readInclude**(*block*)

Creates a sub-instance of *inpRW* to read the \*INCLUDE input file.

The contents of the new input will be placed in the *suboptions* of *block*.

**Parameters**

**block** (*inpKeyword*) – An *inpKeyword* object with *name* = INCLUDE

**\_readManifest**(*block*)

Creates a sub-instance of *inpRW* to read the \*MANIFEST input files.

The contents of the new input will be placed in the *suboptions* of *block*.

**Parameters**

**block** (*inpKeyword*) – An *inpKeyword* object with *name* = MANIFEST

**\_splitKeywords**()

Splits the text string of the input file using the \* and letter characters.

This function splits the text string of the input file using the \* and letter characters. Comment lines (denoted with “\*\*”) will be included with the previous string. Comments prior to the first keyword block will be stored in *\_leadingcomments*.

**\_subParam**(*obj*)

Converts references to a parameter defined in \*PARAMETER to the actual value.

If *obj* is not an instance of *str*, *obj* will be returned untouched. If *obj* is an instance of *str* this function will check if it is a reference to a \*PARAMETER value (i.e. ‘<var1>’). If so, it will retrieve the appropriate value from *pd*. If not, *obj* will be returned.

This function will not modify the original *data* object.

**Parameters**

**obj** (*str* or *inpString*) – *obj* should be a *str* or *inpString* for this function to operate on it.

**Returns**

The type of the returned item is determined by the \*PARAMETER function, or *obj* is returned.

## 12.2.2 inpRW.\_inpW

### Module Contents

This module contains functions for writing a new input file from the information in *keywords*.

#### class inpRW.\_inpW.Write

The *Write* class contains functions that write input files from the parsed data structure.

##### generateInpSum()

Generates the .sum file, which includes the keyword line and *path* for each keyword block.

The file will be written to *\_kwsumName*.

##### writeBlocks(iterable, output=None)

Generates strings from a sequence of *inpKeyword* objects.

This function generates strings from a sequence of *inpKeyword* objects. It will either write the strings to *output*, if *output* is a file, or it will append the strings to *output* if *output* is a list.

This function will call itself recursively to handle suboptions.

##### Parameters

- **iterable** (*list*) – Items in list should be *inpKeyword* blocks.
- **output** (*file object* or *list*) – Output will be written to this object. Defaults to [].

##### Returns

If output is a list, this function will return a list. If output is a file, there is no return.

##### Return type

*list*

##### writeInp(blocks=None, output="")

This function writes out the contents of the input file structure (including any changes) to the file indicated by *output*.

If *blocks* is not specified, every keyword block will be written. If *output* is not specified, the output .inp will be *inpName* + *jobSuffix*. *inpName* will have “.inp” removed before *jobSuffix* is added.

##### Parameters

- **blocks** (*list*) – The list of *inpKeywords* to write to output. Defaults to None, which will write out all keyword blocks.
- **output** (*str*) – The name of the file to which the new input file will be written. Defaults to ‘’, which will write to *inpName* + *jobSuffix*.

## 12.2.3 inpRW.\_inpFind

### Module Contents

This module contains functions for finding information in *keywords*.

#### class inpRW.\_inpFind.Find

The *Find* class contains functions related to finding information from the *inpRW* data structure.

**findItemReferences**(*blockName*, *refNames*)

This function will call the appropriate function from *FindRefs* based on the value of *blockName*. *refNames* will serve as the input to the final function.

This function is mainly meant to be used in *deleteItemReferences()*, where chains of data items and their references need to be deleted. When this happens, a function needs to be called which will search for references to the new keyword block.

**Parameters**

- **blockName** (*str*) – A string which should be the *name* attribute of a keyword block. This function will call `rs1(blockName).title()` to format *blockName* prior to calling the appropriate function in *FindRefs*, so the user just needs to specify the full name of the Abaqus keyword in any formatting, without the `*`.
- **refNames** (*list*) – A sequence of reference items for which to search. These could be strings (i.e. names of sets, surfaces, etc.) or integers (i.e. node labels).

**Returns**

*csid*

**findKeyword**(*keywordName*="", *parameters*=None, *excludeParameters*=None, *data*="", *mode*='all', *printOutput*=True, *parentBlock*="")

Finds keyword blocks in *keywords* using multiple criteria.

This function uses a system of filters to find keyword blocks. It will first create a list of potential blocks by getting blocks that match *keywordName*. It will then try to match using *parameters*, and finally *data*. If one of *keywordName*, *parameters*, or *data* are not specified, they will not be used to filter the search. For example, if we call `inp.findKeyword(keywordName='BOUNDARY')`, this will return all `*BOUNDARY` keyword blocks. If we call `inp.findKeyword(keywordName='BOUNDARY', parameters='AMPLITUDE=amp1')` this will return the `*BOUNDARY` keyword blocks that include `'AMPLITUDE=amp1'` as a parameter.

All inputs to this function are case-insensitive.

*keywordName* can be a string or a list of strings indicating for which keywords to search. Potential blocks will be taken from matching entries in *kmg* instead of looping through every block in *keywords* to find matches.

*parameters* can be a string, a list, or a dictionary of parameters for which to look. The parameters may or may not include values. Specifying *parameters* as a string will be the simplest option for the user. This function will convert *parameters* to a parameter *csid* if it is not already one.

*excludeParameters* can be a string, a list, or a dictionary of parameters. A keyword block must not include a parameter in *excludeParameters* for *findKeyword()* to match the keyword block. The parameters may or may not include values. Specifying *excludeParameters* as a string will be the simplest option for the user. This function will convert *excludeParameters* to a parameter *csid* if it is not already one.

*data* should be a string for which the entirety of the data for the keyword block will be searched, or it can be a list containing parsed data objects. If *data* is a list, the matching will be very strict, so it is best to retrieve the list items from the keyword block's data field directly. Specifying *data* as a list is primarily intended for cases where you have identified a keyword block of interest, but need to perform an operation that moves the keyword block (such as deleting or inserting keywords) and precisely find the same keyword block in its new location.

*mode* should specifically be 'all' (default), 'any', or 'keyandone'. If *mode* == 'all', only keyword blocks that match every input to this function will be returned. If *mode* == 'any', every keyword block that matches one of the inputs will be returned. If *mode* == 'keyandone', every keyword block that matches *keywordName* and at least one other input will be returned. *mode* == 'all' will likely be the most useful.

*printOutput* is a boolean: True (default) or False. If True, the function will print whether it found any keyword blocks matching the input parameters. True is useful if using this function interactively, False is useful if calling this function in batch and you don't want a lot of printed output.

*parentBlock* should be a keyword object that will contain suboptions. This function will search for keywords that match the input parameters, but the potential matches are limited to the suboptions of *parentBlock*. This is very useful for finding a \*BOUNDARY in a specific \*STEP, for example.

---

**Note:** modes 'any', and 'keyandone' have not been extensively tested with the *parentBlock* and list of *keywordName* options, so they might not behave properly.

---

**Warning:** Do not use this function if you need to search for many specific keywords in a large keyword group. For example, you should not use this function to search for 100,000 \*COUPLING blocks with different surface names. Each time this function is called, it will loop through the list of available keywords searching for the appropriate item. If you have a specific search you need to perform multiple times, you should instead create a *csid* using the item of interest as the key and the block as the value. Example: `coupD = csid([ [block.parameter['surface']], [block]] for block in self.kwg['coupling'] ])`

---

**Todo:** The search for data functionality should be augmented to search for data in specific lines or locations in a line to give fewer false positives.

---

### Parameters

- **keywordName** (*str* or *list*) – The name(s) of the keyword(s) for which to search. Defaults to ''.
- **parameters** (*list* or *str* or *dict*) – The parameters and possibly values for which to search. Users are recommended to use a string, as this will be simplest.
- **excludeParameters** (*list* or *str* or *dict*) – The parameters and possibly values which should not be in keywords (i.e. the block will only match if the other parameters are matched and *excludeParameters* is not matched). Users are recommended to use a string, as this will be simplest.
- **data** (*str* or *list*) – A string or list for which to search in any potential block's data.
- **mode** (*str*) – Indicates the matching mode. Can be one of 'all', 'any', or 'keyandone'. Defaults to 'all'
- **printOutput** (*bool*) – Indicates if this function should print out a summary (i.e. how many keyword blocks matched the given criteria). Defaults to True.
- **parentBlock** (*inpKeyword*) – If parentBlock is specified, the search will be restricted to *suboptions* of parentBlock.

### Returns

A list of all matching *inpKeyword* blocks.

### Return type

*list*

### **findKeywordsIgnoreParam**(*kwNameGroup*)

Searches *kwg* for the keywords in *kwNameGroup*, and returns the keyword groups it finds.

**Parameters**

**kwNameGroup** (*list*) – A sequence of strings designating the keyword names of interest.

**Returns**

This returns a list of *keyword groups* as specified by “kwNameGroup”.

**Return type**

*list*

**findStepLocations()**

Creates objects *steps* and *step\_paths*.

If a step does not set the “name” *parameter*, the key for the step will be its index in *kwg['STEP']*.

This function is called by *updateInp()*.

**findValidParentKW(child)**

Prints the keyword names that can be parent blocks for child.

This function searches through all subkeywords for every parent keyword in *\_subBlockKWs*. *\_subBlockKWs* gives you the valid child keywords for a given parent keyword; this function finds the valid parent keywords for a given child keyword.

**Parameters**

**child** (*str*) – The keyword name for which the valid keyword block

**getParentBlock(item, parentKWName="", \_level=1)**

Retrieves the parent block for *item*.

This function will find the parent keyword block for *item*. If *parentKWName* is specified, the function will keep searching up the *path* of *item* until it finds *parentKWName*.

**Parameters**

- **item** (*inpKeyword*, *str*, *list*, or *tuple*) – If *item* is a string, it must be of the form in *path*. Example: 'self.keywords[0].suboptions[1]'. If *item* is a list or tuple, it must be of the form of the *seq* parameter described in *createPathFromSequence()*.
- **parentKWName** (*str*) – Must correspond to a valid Abaqus keyword. Case- and space-insensitive.
- **\_level** (*int*) – Indicates how many times to split *item.path*. Should only be specified by this function.

**Returns**

If a parent block is found that matches the input parameters, an *inpKeyword* is returned. Otherwise, this function returns None.

**Return type**

*inpKeyword* or None

**\_findActiveSystem()**

module: *\_inpFind.py*

Finds which \*SYSTEM is active at all points in the .inp

**Warning:** THIS FUNCTION HAS NOT BEEN TESTED WITH THE inpRW MODULE. IT NEEDS TO BE REWORKED BEFORE IT SHOULD BE USED.

**findCloseNodes**(*nodeGroup1=None, nodeGroup2=None, searchDist=1e-06, \_sliceStart=1*)

This function searches for nodes that are within *searchDist* of each other. It will generate a `cKDTree` from `scipy.spatial`, and the function will return a dictionary using the pairs of nodes which fall within *searchDist* as the key-value pairs.

*nodeGroup1* must be a `TotalMesh` or `Mesh` instance. In most cases the user should use `nd`, which will contain all nodes in the input file.

The nodes in *nodeGroup1* will be used to generate a `cKDTree`, which will be stored to `_tree`. If `_tree` already exists and *nodeGroup1* contains the same number of entities as `_tree`, the existing tree will be used. The `cKDTree` allows for quick nearest-neighbor lookups.

*nodeGroup2* can be specified as a `Mesh` or `TotalMesh` instance. These nodes are the source nodes for which this function will search for close nodes from *nodeGroup1*. If *nodeGroup2* is not specified, it will be identical to *nodeGroup1*. *nodeGroup2* can be a subset of *\*nodeGroup1*; this function will automatically filter out nodes which are only close to themselves. A node from *nodeGroup1* will be considered close to a node from *nodeGroup2* if it falls within *searchDist* of the *nodeGroup2* node. The search is performed using `query_ball_point()`, with *nodeGroup2* being passed to the “x” parameter and *searchDist* being passed to the “r” parameter.

Not specifying *nodeGroup2* will thus look for close nodes over the entire mesh. If the user has an idea of where to find close nodes (i.e. a particular node set), then *nodeGroup2* should be specified to reduce the number of locations this function needs to check.

*searchDist* should be a small float. This defaults to 1e-6.

`_sliceStart = 1` will remove the 0th data column (i.e. remove the node labels). It should not be changed in most cases as this function is written specifically to operate on information stored in `Mesh` or `TotalMesh` instances.

The output dictionary will be key-value pairs of node labels. There will strictly be one value for each key. If *nodeGroup2* is specified, the values of this dictionary will be node labels from *nodeGroup2*, and the keys will be node labels from *nodeGroup1* which are close to each node from *nodeGroup2*. For example, let’s say we have three nodes (1, 2, and 3) which are within *searchDist* of each other (and many other nodes in the mesh), and we want to search for the nodes which are close to node 3. The output of this function would look like the following:

```
{1: 3, 2: 3}
```

If *nodeGroup2* is not specified, the lower node label will always be the value.

#### Parameters

- **nodeGroup1** (`Mesh` or `TotalMesh`) – *nodeGroup1* defines the search domain. In most cases, it should include every node in the input file. Defaults to `nd`, which will include every node.
- **nodeGroup2** (`Mesh` or `TotalMesh`) – *nodeGroup2* defines the nodes for which close nodes should be found. If *nodeGroup2* is specified, the each value in the output dictionary will be a node label from *nodeGroup2*. Defaults to `nd`.
- **searchDist** (`float`) – The distance which defines close nodes. Defaults to 1e-6.
- **\_sliceStart** (`int`) – Used to specify the “column” in a list where the nodal coordinates begin. Since the datalines for a node begin with the node label before the coordinates, `_sliceStart` defaults to 1. Since this function needs to operate on `Mesh` or `TotalMesh` objects, this argument should be left at the default of 1.

#### Returns



This dictionary will always contain a 1:1 mapping of nodelabels which fall within *searchDist*

of each other. If *nodeGroup2* is specified, a nodelabel from *nodeGroup2* will always be the value, otherwise the lower nodelabel will be the value.

#### Return type

dict

**\_findData**(*labelD*, *kwNames*=None, *parameters*=None, *excludeParameters*=None, *mode*='all', *data\_pos*='', *linesToCheck*='', *custom*='', *dataOffset*=None, *dataStep*=None, *lineOffset*=None, *lineStep*=None, *dataGroup*='line', *kwBlocks*=None)

This function will search for references to the keys in *labelD* in the locations specified by the rest of the parameters to this function. It is meant to be called from one of the functions in *FindRefs*.

If *kwBlocks* is specified, *\_findData()* will check the locations indicated by any combination of *data\_pos*, *linesToCheck*, *custom*, *dataOffset*, *dataStep*, *lineOffset*, and *lineStep* for items which are in *labelD*. If any location matches an item in *labelD*, the item in *labelD* will be updated with the path to the location, along with the region of the keyword block the item affects (specified by *dataGroup*). If *kwBlocks* is not specified, *kwNames* must be specified, and *findKeyword()* will be called with arguments *kwNames*, *parameters*, *excludeParameters*, and *mode*. The function will then run as before, but using the resulting keyword blocks from *findKeyword()* instead of those specified by *kwBlocks*.

When using *kwNames*, this function will find the *inpKeyword* blocks to search using this command: ``list(flatten([j for j in self.findKeyword(keywordName=kwNames, parameters=parameters, excludeParameters=excludeParameters, mode=mode, printOutput=False) if j]))`` If this is not sufficient to generate a list of keywords to search, the list should be generated prior to calling *\_findData()* and input using the *kwBlocks* parameter.

#### Parameters

- **labelD** (*csid*) – A csid of the form ``csid([[name, []] for name in names]]``. The items in *names* can be *integers* or *strings*, but the function might not work properly if *labelD* contains both types.
- **kwNames** (*list*) – A sequence containing the keyword names in which to search. Will be passed to *findKeyword()*. Defaults to None. Must be specified if *kwBlocks* is omitted.
- **parameters** – See *findKeyword()*. Defaults to None.
- **excludeParameters** – See *findKeyword()*. Defaults to None.
- **mode** (*str*) – See *findKeyword()*. Defaults to 'all'.
- **data\_pos** (*int* or *str*) – Indicates which positions in the dataline to search for a match. Can be an *int*, or a *str*. If a *str*, can be 'EVEN', 'ODD', or a string indicating a slicing notation. Defaults to '', which will be converted to '[:]' and check every data position. Works in conjunction with *linesToCheck*
- **linesToCheck** (*int* or *str*) – Indicates which datalines to search for a match. Can be an *int*, or a *str*. If a *str*, can be 'EVEN', 'ODD', or a string indicating a slicing notation. Defaults to '', which will be converted to '[:]' and check every dataline. Works in conjunction with *data\_pos*.
- **custom** (*function*) – A custom function to determine which locations should be examined for data matching. This is an alternative to *data\_pos* and *linesToCheck*, and should only be used in the few cases where those options are not sufficient. This function will have access to all the local variables of *\_findData()*, and needs to be written like it's a part of *\_findData()*. See function *externalFieldCustomSub* inside *findElementRefs()* for an example.



- **dataOffset** (*int*) – Used to override the value the function will calculate. They are used in this manner with slicing [Offset : : Step]. They should not be used in most cases, as the appropriate values will be calculated from *data\_pos* and *linesToCheck*, but they are still options for the rare cases the default calculation is incorrect. Defaults to None.
- **dataStep** (*int*) – Used to override the value the function will calculate. They are used in this manner with slicing [Offset : : Step]. They should not be used in most cases, as the appropriate values will be calculated from *data\_pos* and *linesToCheck*, but they are still options for the rare cases the default calculation is incorrect. Defaults to None.
- **lineOffset** (*int*) – Used to override the value the function will calculate. They are used in this manner with slicing [Offset : : Step]. They should not be used in most cases, as the appropriate values will be calculated from *data\_pos* and *linesToCheck*, but they are still options for the rare cases the default calculation is incorrect. Defaults to None.
- **lineStep** (*int*) – Used to override the value the function will calculate. They are used in this manner with slicing [Offset : : Step]. They should not be used in most cases, as the appropriate values will be calculated from *data\_pos* and *linesToCheck*, but they are still options for the rare cases the default calculation is incorrect. Defaults to None.
- **dataGroup** (*str*) – Must be one of ‘line’, ‘multiline’, ‘alldata’, or ‘subline’. Indicates the region of the keyword associated with the reference (i.e. if the reference is deleted, how much of the keyword block must be deleted to still have a valid keyword). ‘line’ corresponds to the entire data line. ‘multiline’ corresponds to multiple keyword lines as indicated by *linesToCheck*. ‘alldata’ corresponds to the entirety of *data*. ‘subline’ corresponds to a region of the dataline as indicated by *data\_pos*. Defaults to ‘line’.
- **kwBlocks** (*list*) – Each item in *kwBlocks* must be an *inpKeyword*. Defaults to None. One of *kwBlocks* or *kwNames* must be specified.

#### **`_findIncludeFileNames()`**

Finds all child input file names in *keywords*.

This function will search for \*INCLUDE, \*MANIFEST, and any keywords in *\_dataKWs* that have the “input” *parameter*. It will populate *includeFileNames* with all file names it finds and *includeBlockPaths* with the *paths* to the blocks that read from child input files.

This function is called by *updateInp()*.

#### **`_findParam(labelD, parameterName, kwNames=None, excludeParameters=None, mode='all', kwBlocks=None)`**

This function will look for keyword blocks that reference items in *labelD*.

It will search *kwBlocks*, or it will search the blocks returned by ``findKeyword(keywordName=kwNames, parameters=parameterName, excludeParameters=excludeParameters, mode=mode, printOutput=False``

#### **Parameters**

- **labelD** (*csid*) – A csid of the form ``csid([[name, []] for name in names])``. The items in *names* can be *integers* or *strings*, but the function might not work properly if *labelD* contains both types.
- **parameterName** (*str*) – A string indicating the parameter name for which to search. This should not include a value. *parameterName* will be passed to *parameters* of *findKeyword()* if *kwBlocks* is not specified. If matching keyword blocks are found, the function will check if the value of ``block.parameter[parameterName]`` is in *labelD*. If so, *labelD[parameterName]* will have `[[path to parameter], {path to block}]` appended.

- **kwNames** (*list*) – A sequence containing the keyword names in which to search. Will be passed to *findKeyword()*. Defaults to None. Must be specified if *kwBlocks* is omitted.
- **excludeParameters** – See *findKeyword()*. Defaults to None.
- **mode** (*str*) – See *findKeyword()*. Defaults to ‘all’.
- **kwBlocks** (*list*) – Each item in *kwBlocks* must be an *inpKeyword*. Defaults to None. Must be specified if *kwNames* is omitted.

**\_getLastBlockPath**(*\_block=""*)

Sets *\_lbp* to the path of the last block in the input file.

**Parameters**

**\_block** (*inpKeyword*) – Only meant to be used when this function calls itself to handle blocks with *suboptions*.

This function is called by *updateInp()*.

## 12.2.4 inpRW.\_inpFindRefs

### Module Contents

This module contains functions for finding references to given names or labels in the input file.

The functions here are not really meant to be used by the end-user; the designed entry point is through *findItemReferences()*. All of the functions in this module are named like the following: “find[Abaquskeywordname]Refs”. [Abaquskeywordname] should be an Abaqus keyword name, with the first word capitalized, no spaces, and all other words lower case. For example, if you wish to find references to a specific \*CONNECTOR BEHAVIOR, you would use findConnectorbehaviorRefs. Using *findItemReferences()* will automatically format the provided keyword name.

These functions are currently somewhat slow. The problem is that they need to search every location in the input file which could reference the desired items. The performance of these functions depends on the number of locations to check far more than the number of items for which to search. For this reason, users should formulate their code so that these functions are called as few times as possible. For example, if the user wishes to search for references to nodes 1-5, they should not call *findNodeRefs()* 5 times, once for each node label. Rather, they should pass in a sequence or set of node labels. The function will check every possible location if the item in each location is in the input sequence.

There are likely errors in these functions, as they are not fully tested and the Abaqus Keywords Guide is not always clear. This class is also not yet complete; the initial focus was on items that can reference node or elements. Please report any problems or new references that need to be addressed to [erik.kane@3ds.com](mailto:erik.kane@3ds.com).

These functions work (at least, once all the bugs are found), but they are slow. They will hopefully be eliminated in the future and replaced with an approach that can track references to specific items instead of needing to search for them.

**class** inpRW.\_inpFindRefs.**FindRefs**

The *FindRefs* class contains functions that find references to specific named entities in the parsed input file structure. For example, *inp.findNodeRefs([1, 2, 3])* will find every location in the parsed input file structure which references node labels 1, 2, and 3.

**findAdaptivemeshcontrolsRefs**(*names*)

findAdaptiveMeshControlsRefs(*names*)

This function searches for keywords that can reference \*ADAPTIVE MESH CONTROLS.

**Parameters**

**names** (*list*) – A sequence of \*ADAPTIVE MESH CONTROLS names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findClearanceRefs**(*names*)

This function searches for keywords that can reference \*CLEARANCE.

**Parameters**

**names** (*list*) – A sequence of \*CLEARANCE names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findConnectorbehaviorRefs**(*names*)

findConnectorBehaviorRefs(names)

This function searches for keywords that can reference \*CONNECTOR BEHAVIOR.

**Parameters**

**names** (*list*) – A sequence of \*CONNECTOR BEHAVIOR names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findContactclearanceRefs**(*names*)

findContactClearanceRefs(names)

This function searches for keywords that can reference \*CONTACT CLEARANCE.

**Parameters**

**names** (*list*) – A sequence of \*CONTACT CLEARANCE names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findContactinitializationdataRefs**(*names*)

findContactInitializationRefs(names)

This function searches for keywords that can reference \*CONTACT INITIALIZATION DATA.

**Parameters**

**names** (*list*) – A sequence of \*CONTACT INITIALIZATION DATA names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findContactpairRefs(names)**

This function searches for keywords that can reference the CPSET parameter of \*CONTACT PAIR.

**Parameters**

**names** (*list*) – A sequence of CPSET names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findDistributionRefs(names)**

This function searches for keywords that can reference \*DISTRIBUTION.

**Parameters**

**names** (*list*) – A sequence of \*DISTRIBUTION names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findDistributiontableRefs(names)**

This function searches for keywords that can reference \*DISTRIBUTION TABLE.

**Parameters**

**names** (*list*) – A sequence of \*DISTRIBUTION TABLE names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findElementprogressiveactivationRefs(names)**

This function searches for keywords that can reference \*ELEMENT PROGRESSIVE ACTIVATION.

**Parameters**

**names** (*list*) – A sequence of \*ELEMENT PROGRESSIVE ACTIVATION names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type***csid***findElementRefs**(*elements*, *mode*='labels')

Finds all references in the input file to each element in *elements*.

**Parameters**

- **elements** (*list*) – A sequence of element labels or element set names (if *mode* == 'set').
- **mode** (*str*) – Indicates which type of items for which the function searches. Valid choices: 'labels': indicates the function will search for element labels (default) 'set': indicates the function will search for element set names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type***csid***findEnrichmentRefs**(*names*)

This function searches for keywords that can reference \*ENRICHMENT.

**Parameters**

**names** (*list*) – A sequence of \*ENRICHMENT names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type***csid***findFastenerpropertyRefs**(*names*)

This function searches for keywords that can reference \*FASTENER PROPERTY.

**Parameters**

**names** (*list*) – A sequence of \*FASTENER PROPERTY names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type***csid***findFieldmappercontrolsRefs**(*names*)

This function searches for keywords that can reference \*FIELD MAPPER CONTROLS.

**Parameters**

**names** (*list*) – A sequence of \*FIELD MAPPER CONTROLS names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type***csid***findFluidbehaviorRefs**(*names*)

This function searches for keywords that can reference \*FLUID BEHAVIOR.

**Parameters**

**names** (*list*) – A sequence of \*FLUID BEHAVIOR names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which  
**references**  
the name and the region of the keyword effected by this reference.

**Return type***csid***findGasketbehaviorRefs**(*names*)

This function searches for keywords that can reference \*GASKET BEHAVIOR.

**Parameters**

**names** (*list*) – A sequence of \*GASKET BEHAVIOR names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which  
**references**  
the name and the region of the keyword effected by this reference.

**Return type***csid***findImpedancepropertyRefs**(*names*)

This function searches for keywords that can reference \*IMPEDANCE PROPERTY.

**Parameters**

**names** (*list*) – A sequence of \*IMPEDANCE PROPERTY names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which  
**references**  
the name and the region of the keyword effected by this reference.

**Return type***csid***findIncidentwaveinteractionpropertyRefs**(*names*)

This function searches for keywords that can reference \*INCIDENT WAVE INTERACTION PROPERTY.

**Parameters**

**names** (*list*) – A sequence of \*INCIDENT WAVE INTERACTION PROPERTY names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which  
**references**  
the name and the region of the keyword effected by this reference.

**Return type***csid*

**findIncidentwavepropertyRefs**(*names*)

This function searches for keywords that can reference \*INCIDENT WAVE PROPERTY.

**Parameters**

**names** (*list*) – A sequence of \*INCIDENT WAVE PROPERTY names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which  
**references**  
the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findIntegratedoutputsectionRefs**(*names*)

This function searches for keywords that can reference \*INTEGRATED OUTPUT SECTION.

**Parameters**

**names** (*list*) – A sequence of \*INTEGRATED OUTPUT SECTION names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which  
**references**  
the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findNodeRefs**(*nodes*, *mode*='labels')

Finds all references in the input file to each node in nodes.

**Parameters**

- **nodes** (*list*) – A sequence of node labels or node set names (if *mode* == 'set').
- **mode** (*str*) – Indicates which type of items for which the function searches. Valid choices:  
'labels': indicates the function will search for node labels (default) 'set': indicates the function will search for node set names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which  
**references**  
the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findOrientationRefs**(*names*)

This function searches for keywords that can reference \*ORIENTATION.

**Parameters**

**names** (*list*) – A sequence of \*ORIENTATION names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which  
**references**  
the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findPeriodicmediaRefs**(*names*)

This function searches for keywords that can reference \*PERIODIC MEDIA.

**Parameters**

**names** (*list*) – A sequence of \*PERIODIC MEDIA names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findRebarlayerRefs**(*names*)

This function searches for keywords that can reference \*REBAR LAYER.

**Parameters**

**names** (*list*) – A sequence of \*REBAR LAYER names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findRebarRefs**(*names*)

This function searches for keywords that can reference \*REBAR.

**Parameters**

**names** (*list*) – A sequence of \*REBAR names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findRigidbodyRefs**(*names*)

This function searches for keywords that can reference \*RIGID BODY.

**Parameters**

**names** (*list*) – A sequence of \*RIGID BODY names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findSectioncontrolsRefs**(*names*)

This function searches for keywords that can reference \*SECTION CONTROLS.

**Parameters**

**names** (*list*) – A sequence of \*SECTION CONTROLS names.



**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findSubstructureloadcaseRefs**(*names*)

This function searches for keywords that can reference \*SUBSTRUCTURE LOADCASE.

**Parameters**

**names** (*list*) – A sequence of \*SUBSTRUCTURE LOADCASE names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findSurfaceRefs**(*names*)

This function searches for keywords that can reference \*SURFACE.

**Parameters**

**names** (*list*) – A sequence of \*SURFACE names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findSurfaceinteractionRefs**(*names*)

This function searches for keywords that can reference \*SURFACE INTERACTION.

**Parameters**

**names** (*list*) – A sequence of \*SURFACE INTERACTION names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type**

*csid*

**findSurfacepropertyRefs**(*names*)

This function searches for keywords that can reference \*SURFACE PROPERTY.

**Parameters**

**names** (*list*) – A sequence of \*SURFACE PROPERTY names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type***csid***findSurfacesmoothingRefs**(*names*)

This function searches for keywords that can reference \*SURFACE SMOOTHING.

**Parameters**

**names** (*list*) – A sequence of \*SURFACE SMOOTHING names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type***csid***findTracerparticleRefs**(*names*)

This function searches for keywords that can reference \*TRACER PARTICLE.

**Parameters**

**names** (*list*) – A sequence of \*TRACER PARTICLE names.

**Returns**

Uses names as the keys, each value will be a list containing the exact location which references the name and the region of the keyword effected by this reference.

**Return type***csid*

## 12.2.5 inpRW.\_inpMod

### Module Contents

This module contains functions for modifying the data in *keywords*.

**class** inpRW.\_inpMod.**Mod**

The *Mod* class contains functions modify data in the parsed input file structure.

**calculateCentroid**(*elementID*)

This function gets the nodal coordinates of an element and calls *centroid.averageVertices()*.

**Parameters**

**elementID** (*int* or *inpInt*) – The label of an element.

**Returns**

A numpy array containing the coordinates of the element centroid.

**Return type***numpy.ndarray***convertOPnewToOPmod**(*startStep=0, finishStep=None, bStep=None*)

Converts keyword parameter OP=NEW to OP=MOD or OP=REPLACE and removes unnecessary keywords from steps.

This function will first identify a baseStep from which to compare all steps between and including *startStep* and *finishStep*. baseStep can be specified via the *bStep* parameter, or it will be the 0th step. The function will

first identify the keyword blocks in `baseStep` which use the “OP” parameter (the list of potential keywords is stored in `_opKeywords`). This will include implied “OP” parameters.

#### Parameters

- **startStep** (*int* or *str*) – The beginning step for which to convert “OP” parameters. Defaults to 0.
- **finishStep** (*int* or *str*) – The last step for which to convert “OP” parameters. If `None`, all steps after and including `startStep` will be converted. Defaults to `None`.
- **bStep** (*int* or *str*) – The index or name of the step in `kwg['Step']` from which the other steps will be compared. If `bStep = None`, the 0th step will be `baseStep`. Defaults to `None`.

**Warning:** This function should not be used. It needs to be reworked, as it does not properly deactivate step features that are inactive in subsequent steps.

---

**Todo:** Read Boundary Conditions in model data as the initial `baseStep`.

---

**createManifest** (*lastBaseStep=None, stepGroupingIn=None, parameterString=""*)

Creates a `*MANIFEST inpRW` instance and places the steps in the original file into sub-input files.

This function will reorganize the steps in the input file per the user’s specification. The new `*MANIFEST` input file will look like the following:

```
*MANIFEST,BASE STATE=YES, EVOLUTION TYPE=HISTORY
model_and_basesteps.inp
stepGrouping1.inp
stepGrouping2.inp
stepGrouping3.inp
```

The first input file will contain the model data and any common base steps for the subsequent simulations. All subsequent input files contain independent simulation histories. See that Abaqus documentation for more information on `*MANIFEST`.

*lastBaseStep* is the last step in the input file that will be considered a base step for the analysis. It can be a 0-based integer corresponding to the step number, or it can be a case-insensitive string corresponding to a step name. If *lastBaseStep* is not specified, all steps in the simulation will be used in nonlinear load cases.

*stepGroupingIn* is a sequence of sequences, the first item is the input file name, and the second is a sequence of the steps to associate with that input file. The sequence can use step names or integers.

For example:

```
stepGroupingIn = [ ['LoadA.inp', ['Step-1', 'Step-2']], ['LoadB.inp', [3, 4]] ]
```

If *stepGroupingIn* is not specified, all steps after *lastBaseStep* will be independent nonlinear load cases. There is no limit on how steps can be ordered, reused, etc. However, it is the analyst’s responsibility to build a working step order, as this function does not check the validity of the simulation.

*parameterString* is a string that allows the user to specify the parameters for `*MANIFEST`. Specify them as you would on the Abaqus keyword line. The “BASE STATE” parameter will be set based on the *lastBaseStep* input. “EVOLUTION TYPE=HISTORY” is automatically included and should not be specified, as this is currently (as of Abaqus 2022 GA) the only valid option.

Example:

```
parameterString = 'MODEL CHANGE, RESULTS=new'
```

### Parameters

- **lastBaseStep** (*int* or *str*) – Indicates which step should be considered the last base step for the \*MANIFEST steps. Use 0-based indices.
- **stepGroupingIn** (*list*) – A Sequence of sequences, the first item is the input file name, and the second is a sequence of the steps to associate with that input file. The sequence can use step names or integers.
- **parameterString** (*str*) – Specifies additional parameters for the keyword line. The “BASE STATE” and “EVOLUTION TYPE” parameters will automatically be included.

### Returns

A new *inpRW* instance reorganized as a \*MANIFEST input file.

### Return type

*inpRW*

**deleteItemReferences**(*labelD*, *inputType*='other', *deleteModCouplings*=False, *delNodesFromElsLevel1*=False, *\_level*=0, *\_limit*=100)

This function will delete the item references in *labelD* throughout the input file. If an item that is deleted also deletes a *named reference*, this function will also find the references to those new items and delete them.

A thorough explanation of the *delNodesFromElsLevel1* option is needed.

The user has a set of nodes to delete. The user generates *labelD* with “labelD = find-NodeRefs(nlabels)”. He then deletes the nodes and all references to the nodes (contained in *labelD*) with `deleteItemReferences(labelD, 'node')`. `deleteItemReferences()` will call itself to handle chained item references, and it will specify the *\_level* parameter to track how many levels deep it is. When deleting nodes, `deleteItemReferences()` will by default delete the nodes and their references, which will include elements. It deletes the elements by calling `deleteItemReferences(elementD, 'element', _level=1)`, which also handles deleting all references to the elements.

`deleteItemReferences(elementD, 'element', _level=0)` behaves a bit differently. When elements are deleted, any nodes that were formally parts of elements but are no longer referenced by any elements will be deleted. This does not happen with `deleteItemReferences(elementD, 'element', _level=1)` unless *delNodesFromElsLevel1*=True is also specified.

This option has some niche use cases (for example, deleting fastener constructs of the form mesh - coupling - solid elements representing weld - coupling - mesh). In most cases, leaving *delNodesFromElsLevel1*=False (default) is the desired choice.

### Parameters

- **labelD** (*csid*) – Contains the mapping between the item references (the keys), and the location of the reference along with the region that should be deleted if the item is deleted (the value). *labelD* should be the return *csid* from a function in *FindRefs*.
- **inputType** (*str*) – Indicates to what *labelD* refers. Valid options are ‘node’, ‘element’, and ‘other’. ‘node’ will handle node label and node set references, ‘element’ will handle element label and element set references, and ‘other’ handles all other cases. Defaults to ‘other’.

- **deleteModCouplings** (*bool*) – If True, all \*COUPLING, \*KINEMATIC COUPLING, and \*DISTRIBUTING COUPLING keywords with modified application regions (\*SURFACE, \*NSET, \*ELSET) will be deleted. Defaults to False.
- **delNodesFromElsLevel1** (*bool*) – If True, nodes can be deleted when element references are deleted in `_level=1`. See detailed explanation above. Defaults to False.
- **\_level** (*int*) – Tracks when `deleteItemReferences()` calls itself to handle certain reference chains. The user should not need to specify this. Defaults to 0.
- **\_limit** (*int*) – A limit on how many iterations this function will perform when trying to identify and delete chains of references. This check is performed in a while loop, which could create an infinite loop if something unexpected happens, hence the limit. The iterations referred to is something like the following: delete a node -> delete an element -> delete an element set -> delete a surface -> delete a contact pair. This example had 5 iterations from deleting nodes to deleting a contact pair using an element based surface. The default value of 100 should be much higher than necessary. Defaults to 100.

**deleteKeyword**(*path=""*, *positionl=None*, *updateInpStructure=True*, *printstatus=True*)

Deletes the keyword block at the indicated location and updates the data structure.

Either *path* or *positionl* is required. Set *updateInpStructure* to False if you need to delete many keyword blocks from a large input file, as those update operations can be slow. If *updateInpStructure* == True, this function will call `updateInp()`, but only update *keyword block* paths after (and including) the position of the deleted block, and it will not call `_groupKeywordBlocks()`, as this function will automatically delete the entry from *kwg*.

**Note:** When deleting a list of keywords, deleting one keyword will change the indices of all subsequent keywords. However, the list of *inpKeyword.paths* or position indices of keywords to delete will not update automatically. For this reason, the user should always sort the list of keywords to delete so that they are deleted from the end of the .inp towards the front. Deleting a keyword object at the end of the list will not affect the indices of keywords prior to it. The sorting is more easily handled when using *positionl* (which is a list of integers) than with *path*, which is a list of strings. If the input file was parsed with *organize=True*, the user will need to perform a multi-level sort to account for the keyword and suboption indices. Use `nestedSort()` for this task. Example:

```
oldBlockPaths = list(flatten(oldBlockPaths))
#gets a flattened list of keyword paths to delete

oldBlockPositionls = [self.parsePath(i)[1:] for i in oldBlockPaths]
#gets the positionl from each path i.e. [1,[0]] will be keyword block 1,
↳suboption 0

oldBlockPositionls1 = nestedSort(iterable=oldBlockPositionls, reverse=True)
#sorts oldBlockPositionls in reverse order. This will account for nested
↳suboptions.
```

Alternately, the user can account for the change of indices when deleting keywords. However, sorting and reversing the list of keywords to delete should be simpler in most cases.

### Parameters

- **path** (*str*) – A string indicating the path to the *inpKeyword* to delete. Defaults to ‘’.

- **positionl** (*list*) – A sequence of sequences with the indices to the *inpKeyword* block to delete. A path string will be generated from this via *createPathFromSequence()*. Defaults to None.
- **updateInpStructure** (*bool*) – If True, will call *updateInp()* after deleting the keyword block. Defaults to True.
- **printstatus** (*bool*) – If True, print which keyword block is deleted. Defaults to True.

**getLabelsFromSet** (*blocks*)

Returns a flat list of all the labels from the set defined in *block*.

**Parameters**

**blocks** (*list*) – A list of \*NSET or \*ELSET keyword blocks.

**Returns**

A flat list containing the labels in the sets. If more than one set is passed to this function, the output list will be sorted and contain only unique entries. If only one set is used, the output will not be sorted, and any duplicate entries will not be removed.

**Return type**

*list*

**insertKeyword** (*obj*, *path=""*, *positionl=None*, *updateInpStructure=True*)

Inserts a new keyword object in the *inpKeywordSequence* at the location specified by *path* or *positionl*.

*obj* is required, and either *path* or *positionl* is required. Set *updateInpStructure* to False if you need to insert many keyword blocks into a large input file, as those update operations can be slow. If *updateInpStructure* == True, this function will call *updateInp()*, but only update *keyword block* paths after (and including) the position of the inserted block. *kwg* will only have the new block added.

---

**Note:** Inserting multiple keywords should be done in reverse order. See the note in *deleteKeyword()* for a detailed explanation.

---

**Parameters**

- **obj** (*inpKeyword*) – The new keyword block to insert.
- **path** (*str*) – A string indicating the path to the *inpKeyword* to delete. Defaults to ‘’.
- **positionl** (*list*) – A sequence of sequences with the indices to the *inpKeyword* block to delete. A path string will be generated from this via *createPathFromSequence()*. Defaults to None.
- **updateInpStructure** (*bool*) – If True, will call *updateInp()* after deleting the keyword block. Defaults to True.

**mergeNodes** (*dupNodeDict*)

Merges pairs of nodes into one node. The node labels referred to by the keys in *dupNodeDict* will be replaced with the node labels in values.

*dupNodeDict* should be a one-to-one mapping of node labels. The key will be replaced by the value. In most cases, this dictionary should be created by using *findCloseNodes()*.

For every pair of nodes in *dupNodeDict*, this function will first find every element connected to the old node (the key). Each of these elements will have their references to the node label in key replaced with the node label in value. Then, the old node will have its *mesh.Node.elements* attribute cleared. Finally, *deleteItemReferences()* will be called to delete the old nodes and remove all remaining references to

them. Thus, the only locations in the input file where the old node labels will be replaced with the new node labels is in \*ELEMENT keyword blocks; all other references will simply be deleted.

#### Parameters

**dupNodeDict** (*dict*) – A dictionary which maps the node label to be removed with the node label with which it should be replaced.

#### **reduceStepSubKWs** (*removeComments=False*)

Consolidates keywords in a particular step that can have the “OP” parameter.

This function can also group some sub-keywords if the parent keywords are identical. This function is limited to keyword blocks in 3 different groupings: *\_mergeDatalineKeywordsOP*, *\_appendDatalineKeywordsOP*, and *\_dofXtoYDatalineKeywordsOP*.

If a keyword block to be removed has comments in its data, those comments will be included with the consolidated keyword block if *removeComments=False*. They will be immediately after the associated dataline if the keyword is in *\_appendDatalineKeywordsOP*, or they will all be appended to the end of the data if the keyword is in *\_dofXtoYDatalineKeywordsOP*.

Example:

```
*BOUNDARY, OP=NEW, ORIENTATION="Orientation48"
"Fixed Displacement24", 2, 2, 0.
*BOUNDARY, OP=NEW, ORIENTATION="Orientation48"
"Fixed Displacement24", 3, 3, 0.
*BOUNDARY, OP=NEW, ORIENTATION="Orientation48"
"Fixed Displacement24", 4, 4, 0.
*OUTPUT, FIELD, FREQUENCY=1
*ELEMENT OUTPUT
S
*OUTPUT, FIELD, FREQUENCY=1
*ELEMENT OUTPUT
E
*OUTPUT, FIELD, FREQUENCY=1
*ELEMENT OUTPUT
PE
*OUTPUT, FIELD, FREQUENCY=1
*ELEMENT OUTPUT
PEEQ
```

will become:

```
*BOUNDARY, OP=NEW, ORIENTATION="Orientation48"
"Fixed Displacement24", 2, 4, 0.
*OUTPUT, FIELD, FREQUENCY=1
*ELEMENT OUTPUT
S, E, PE, PEEQ
```

#### Parameters

**removeComments** (*bool*) – If True, any comments associated with keyword blocks that are now consolidated will be deleted. Defaults to False.

---

**Todo:** Add support for keywords in *\_mergeDatalineKeywordsOP*.

---

**removeGenerate**(*generateBlocks=None*)

Removes the GENERATE parameter from keyword blocks and expands the datalines.

This function will loop through the keyword blocks as specified in *generateBlocks* or it will search the entire *inpKeywordSequence* for the appropriate blocks. It will remove the “GENERATE” parameter from blocks such as \*NSET, and fully expand the datalines. It will ignore \*TIME POINTS, \*NODAL THICKNESS, and \*NODAL ENERGY RATE keywords if *generateBlocks* == None.

**Parameters**

**generateBlocks** (*list*) – A list containing *inpKeyword* blocks. Defaults to None.

---

**Todo:** Test on latest version of inpRW.

---

**replaceKeyword**(*obj, path="", positionl=None, updateInpStructure=True*)

Deletes the keyword at the position designated by *path* or *positionl*, and then inserts the new keyword object at the same location.

*obj* is required, and either *path* or *positionl* is required. Set *updateInpStructure* to False if you need to insert many keyword blocks into a large input file, as those update operations can be slow. If *updateInpStructure* == True, this function will call *updateInp()*, but only update *keyword block* paths after (and including) the position of the inserted block. *kwg* will only have the old block deleted and the new block added.

**Parameters**

- **obj** (*inpKeyword*) – The new keyword block to insert.
- **path** (*str*) – A string indicating the path to the *inpKeyword* to delete. Defaults to ‘’.
- **positionl** (*list*) – A sequence of sequences with the indices to the *inpKeyword* block to delete. A path string will be generated from this via *createPathFromSequence()*. Defaults to None.
- **updateInpStructure** (*bool*) – If True, will call *updateInp()* after deleting the keyword block. Defaults to True.

**updateKTRIDD**(*keywordNames*)

Short for “update Keywords To Remove If Data Deleted”, this function will update *ktridd*.

**Parameters**

**keywordNames** (*list*) – A sequence of keyword names.

**updatePKTRICD**(*mapping*)

Updates *pktricd* with the items in mapping.

Each key in *pktricd* will be an *inpKeyword* block. The value will be parent keyword name which should be deleted if the *inpKeyword* contained in the key is deleted.

**Parameters**

**mapping** (*list*) – A sequence of sequences of the form [[[*inpKeyword*, *inpKeyword*, ...], parentKWName (*str*)]].

**\_consolidateDoFXtoYDatalines**(*datalist, removeComments=False*)

Examines *datalist* to determine the DOF ranges that are constrained. Returns a consolidated list.

This should be specific to the \*ADAPTIVE MESH CONSTRAINT and \*BOUNDARY (with certain dataline lengths) keywords. The function will group the datalines by application region and by applied load, and will consolidate the applicable DOFs. This will return new dataline(s) as applicable.

The first grouping is via the application region, which is the 0th field in the dataline. This should be a node label or node set name. This function just uses the item in the field as the key; so if one \*BOUNDARY



applies to node 1, and another applies to nset1, which contains only node 1, the BCs will not be consolidated together.

The items in datalist should all correspond to identical keyword lines, as it would be inappropriate to group items that use different parameters. That must be handled before calling this function. Thus, the preferred method for calling this function is through `reduceStepSubKWs()`, as this will perform the necessary operations prior to calling this function.

Example:

```
In: datalist = [{"Fixed Displacement24", 2, 2, 0.}, {"Fixed Displacement24", 3, 3, 0.}, {"Fixed Displacement24", 4, 4, 0.}]
In: self._consolidateDofXtoYdatalines(datalist)
Out: [{"Fixed Displacement24", 2, 4, 0.}]
```

#### Parameters

- **datalist** (*list*) – This should be a sequence of parsed datalines, not a list of strings.
- **removeComments** (*bool*) – If True, comments included in datalist will be discarded. If False, they will be appended to the output.

#### Returns

A list of the consolidated datalines.

#### Return type

*list*

#### `_convertSystem(node)`

convertSystem(node) module: `_inpMod.py`

**Warning:** THIS FUNCTION HAS NOT BEEN TESTED WITH THE inpRW MODULE. IT NEEDS TO BE REWORKED BEFORE IT SHOULD BE USED.

Pass in node data as (X, Y, Z, cos1, cos2, cos3, ActiveSystem tuple):

Example:

```
(1, 10.0, 0.0, 20.0, 0.0, 0.0, 0.0, ((0.0, 0.0, 0.0, 1.0, 0.0, 0.0),))
```

Returns the X,Y,Z coordinates of the node in the global CSYS.

---

**Todo:** Convert to inpRW.

---

#### `_isolateNodesToKeep(dup_node_dict, nodesToKeep)`

**Warning:** THIS FUNCTION HAS NOT BEEN TESTED WITH THE inpRW MODULE. IT NEEDS TO BE REWORKED BEFORE IT SHOULD BE USED.

`dup_node_dict` is the duplicate nodes dictionary produced by `_findCloseNodes`, and `nodesToKeep` is a sequence of nodes. This function will remove all nodes in `nodesToKeep` from `dup_node_dict`.

**`_removeSystem()`**

Finds the active system at all points of the input file, converts all node definitions to global coordinates, then deletes the \*SYSTEM keywords.

**Warning:** THIS FUNCTION HAS NOT BEEN TESTED WITH THE inpRW MODULE. IT NEEDS TO BE REWORKED BEFORE IT SHOULD BE USED.

**`_replaceStarParameter()`**

module: `_inpMod.py`

This function will delete \*PARAMETER keyword lines, and substitute the appropriate values into the input file.

**Warning:** THIS FUNCTION HAS NOT BEEN TESTED WITH THE inpRW MODULE. IT NEEDS TO BE REWORKED BEFORE IT SHOULD BE USED.

## 12.2.6 `inpRW._importedModules`

This module contains all the module imports used by the submodules of *inpRW*. No module that is imported in *\_importedModules* should import *\_importedModules*, or circular import errors will likely occur.

## 12.2.7 `inpRW._inpCustom`

### Module Contents

**`class inpRW._inpCustom.Custom`**

This class is intentionally blank, as it allows end users without access to *inpRW* source code a method to extend *inpRW*. This class is automatically imported as part of *inpRW*. Scripters only need to define new functions here and optionally delete the “custom” class variable. `self` can be used to refer to the *inpRW* instance.

## 13.1 Module Contents

This module contains functions with different methods of calculating centroids.

`centroid.averageVertices(nodeCoords)`

Estimates the centroid of *nodeCoords* by simply averaging the coordinates.

**Parameters**

**nodeCoords** (*list*) – A list of lists containing the nodal coordinates to be averaged.

**Returns**

A numpy array containing the coordinates of the element centroid.

**Return type**

`numpy.ndarray`



## 14.1 Module Contents

In addition, it sets several groupings of keyword names and other similar items that are constant and need to be referenced throughout *inpRW*. See [How do I share global variables across modules?](#) for more information.

### `config._slash`

Stores slash or backslash depending on the OS; this will be inserted into all file paths and makes the script OS independent.

**Type**  
*str*

### `config._openEncoding`

Indicates the encoding used to open and write text files. Can be changed if needed before the user calls *parse()*.

**Type**  
*str*

### `config._supFilePath`

Supplemental file path, specifies the location of the additional files *inpRW* needs to read from. Defaults to the directory of *inpRW* and should not be changed.

**Type**  
*str*

### `config._elementTypeDictionary`

A case- and space-insensitive dictionary that contains information for each element type. The element type labels are the keys, and *elType* instances are the values.

**Type**  
*csid*

### `config._mergeDatelineKeywordsOP`

Keywords whose data items can be consolidated by merging them into new datalines with up to X items per line.

**Type**  
*dict*

### `config._appendDatelineKeywordsOP`

Keywords whose datalines can be consolidated by appending them one after another.

**Type**  
*set*

**config.\_dofXtoYDataLineKeywordsOP**

Keywords that can be consolidated as DoF X to DoF Y. Boundary is not included here as it requires special handling.

**Type**  
set

**config.\_opKeywords**

Keywords that can have the OP parameter.

**Type**  
set

**config.\_subBlockKWs**

Dictionary that uses a keyword name as the key, and a list of all keywords that are valid suboptions for that block as the value.

**Type**  
*csid*

**config.\_allKwNames**

Contains all keyword names. Currently not used.

**Type**  
set

**config.\_EoFKWs**

Keywords that read until the end of a file.

**Type**  
set

**config.\_dataKWs**

Keywords that can read their data from another file.

**Type**  
set

**config.\_EndKWs**

Keywords that have an associated \*END KEYWORD.

**Type**  
set

**config.\_generalSteps**

Keywords that are general procedures.

**Type**  
set

**config.\_perturbationSteps**

Keywords that are linear perturbation procedures.

**Type**  
set

**config.\_keywordsDelayPlacingData** = {csiKeyString('element'): 2,  
csiKeyString('parameter'): 0}

Keywords in this list will not have their data parsed until loop X. This is for specific keywords that need to operate on other parsed data; for example, \*ELEMENT needs *nd* to be fully populated, which requires waiting until all node blocks have been parsed. If a keyword is not in this dictionary, it will be parsed at 1.

**Type**  
*csid*

`config._maxParsingLoops = 2`

The maximum number of loops to parse keyword data. This should match the highest number in the *\_keywordsDelayPlacingData*. The initial loop will use an integer value of 0.

**Type**  
*int*





## 15.1 Module Contents

This module contains compiled regular expressions which will be used throughout *inpRW*. In addition, it sets several groupings of keyword names and other similar items that are constant and need to be referenced throughout *inpRW*. See [How do I share global variables across modules?](#) for more information.

```
config_re.re1 = re.compile('\S')
```

Matches a non-whitespace character

```
config_re.re2 = re.compile('".*?"')
```

Matches everything between quotes, non-greedy

```
config_re.re3 = re.compile('(?(<=\\<)\\w*(?=\\>))')
```

Matches Abaqus PARAMETER reference (i.e. <variable1>)

```
config_re.re4 = re.compile('(?(<=\\S)\\s+(?=\\S)')
```

Matches all whitespace between non-whitespace

```
config_re.re5 = re.compile('\\S+')
```

Matches 1 or more non-whitespace character

```
config_re.re7 = re.compile('[de]', re.IGNORECASE)
```

Matches scientific notation symbol d or e, case-insensitive

```
config_re.re8 = re.compile('[dDeE]')
```

Matches scientific notation symbol d or e, case-insensitive

```
config_re.re9 = re.compile('\\d')
```

Matches one numeric digit

```
config_re.re10 = re.compile('[dD]')
```

Matches d or D

```
config_re.re11 = re.compile('(?(<=-) +')
```

Matches 1 or more spaces if they are preceded by “-”

```
config_re.re12 = re.compile('(?![a-zA-Z])\\-?[1-9]+[0-9]*(?![a-zA-Z])')
```

Matches a valid Python integer, which must not have any letter characters, and must start with a non-zero digit

```
config_re.re13 = re.compile('(?(<=\\d)[dD](?=[-\\+])\\d+')
```

Matches a scientific notation number with d or D as the exponent symbol. Not currently used

```
config_re.re14 = re.compile('\\r*\\n')
```

Matches 0 or more carriage returns and then a new line

```
config_re.re15 = re.compile('\\r*\\n(?:[ \\t]*\\*[a-zA-Z])')
```

Matches the whitespace immediately prior to the start of an Abaqus keyword (one “\*” and a letter)

```
config_re.re16 = re.compile('[ \\t]*\\*[aA-ZZ]')
```

Matches the whitespace prior to an Abaqus Keyword, and the start of the Abaqus keyword “\*” and one letter. Single line only.

```
config_re.re17 = re.compile('\\r*\\n(?:!=\\*\\*)')
```

Matches any number of carriage returns and a new line, as long as they are not followed by an Abaqus comment “\*\*”

```
config_re.re18 = re.compile('([ \\t]+)(?=\\*[a-zA-Z])')
```

Matches any whitespace prior to the \* in a keyword line

```
config_re.re19 = re.compile('(?(=[de])[+-]?\\d*\\s*', re.IGNORECASE)
```

Matches the exponent value of a number and trailing whitespace, not including the exponent symbol. For example, if we perform `re19.search('1.25 e-2')`, the result will be -2.

```
config_re.re21 = re.compile('(?(=.keywords\\[\\]-?\\d+(?=\\[\\]))')
```

Matches the indices of the keywords portion of an *path*.

```
config_re.re22 = re.compile('(?(=.suboptions\\[\\]-?\\d+(?=\\[\\]))')
```

Matches the indices of the suboptions portion of an *path*.

```
config_re.re23 = re.compile('(?(=.data\\[\\]-?\\d+(?=\\[\\]))')
```

Matches the indices of the data portion of an *path*.

```
config_re.re24 = re.compile('(?(=\\[\\]-?\\d+(?=\\[\\]))')
```

Matches indices between brackets (i.e. [“100”])

```
config_re.re25 = re.compile('.*(?:\\*[aA-ZZ])')
```

Matches whatever is on the line before the start of a keyword (\*A). Used only to find leading comments before first keyword.

```
config_re.re26 = re.compile('\\\\s*(?!\\\\S)\\\\Z')
```

Searches for trailing comma and optionally whitespace at the end of a string. Used to check if an element definition has been completed.

## 16.1 Module Contents

This module implements a few classes that together enable a custom case- and space-insensitive dictionary class.

**class** `csid.csid(data=None)`

Bases: `dict`

This is a custom class that overrides lookup functions of dictionary classes. It makes their keys case-insensitive and space-insensitive.

Since this class inherits from the dictionary type, it behaves almost exactly like a dictionary. The main difference is the keys to the dictionary. Any operation with a `csid` instance will automatically generate and use a `csiKeyString` for the key if the key is a string type, or a `csiKeyDecimal` if the key is a `float` (the hash of a float is not consistent with the hash of a `Decimal`). Otherwise, the object itself will be used as the key, as other object types should not support spaces nor capitalization.

Constructing a `csid` instance uses the same syntax as constructing a dictionary, or you can convert an existing dictionary to a `csid` via `csid(dict)`.

**\_\_init\_\_**(`data=None`)

Creates the dictionary. Uses the same construction methods as `dict`.

If `data` is specified, each key in `data` will be converted to a `csiKeyDecimal` or `csiKeyString` prior to creating the entry in the `csid`.

Example usage:

```
>>> from csid import csid
>>> d = csid()
>>> d[' Nodeset-1'] = 'test'
>>> d[' Nodeset-2'] = 'test2'
>>> print(d)
{
  ' Nodeset-1': 'test',
  ' Nodeset-2': 'test2'
}
```

We can also populate the `csid` using existing data, providing it's in a format which can be used to create a `dict`:

```
>>> l = [['Nodeset-1', 1], ['Nodeset-2', 2], ['Nodeset-3', 3]]
>>> csid(l)
```

(continues on next page)

(continued from previous page)

```
csid(csiKeyString('Nodeset-1'): 1, csiKeyString('Nodeset-2'): 2, csiKeyString(
↪ 'Nodeset-3'): 3)
```

If *data* is not formatted properly, a `TypeError` or `ValueError` will be raised:

```
>>> csid(['Nodeset-1', 'Nodeset-2', 'NodeSet-3'], [1, 2, 3])
Traceback (most recent call last):
...
ValueError: too many values to unpack (expected 2)
>>> csid([1,2,3])
Traceback (most recent call last):
...
TypeError: cannot unpack non-iterable int object
```

Please note that if two keys in *data* are identical once they have been made lowercase and had spaces removed, the second entry will overwrite the first one. Example:

```
>>> csid(['Nodeset-1', 1], ['NODESET-1', 2], ['Nodeset-3', 3])
csid(csiKeyString('Nodeset-1'): 2, csiKeyString('Nodeset-3'): 3)
```

### Parameters

**data** (*iterable*) – Defaults to None.

### `get(key, default=None)`

Converts *key* to a *csiKeyString* or *csiKeyDecimal* if necessary, and then retrieves the entry in the *csid* instance using `get()`.

If *key* is not in the *csid*, *default* will be returned.

Example usage:

```
>>> from csid import csid
>>> d = csid([[100, 1]])
>>> d.get(100)
1
```

If *key* is not in the *csid*, *default* is returned:

```
>>> print(d.get(200))
None
```

### Parameters

**key** (*str*, *float*, *Decimal*) – The key to the item to retrieve from the dictionary. *key* will be converted to a *csiKeyString* or *csiKeyDecimal* if needed.

### Returns

The value or None.

### `mergeSubItems(other)`

Merges the sub-items of *other* into the appropriate item in *d* (the *csid* instance).

For *k* and *v* in *other.items()*, if *d[k]* is not defined, set *d[k]* to *v*. If *d[k]* is a list, perform *d[k].extend(v)*. If *d[k]* is a dictionary, perform *d[k].update(v)*.

This function is meant for cases where `update()` won't be sufficient. For example, if `d` and `other` both have a key `k`, the content of each is a list, and you wish the value associated with `k` to be a list containing the contents of both entries.

Example usage:

```
>>> from csid import csid
>>> d = csid([[1, [1,2,3]]])
>>> d2 = csid([[1, [4,5,6]]])
>>> d.mergeSubItems(d2)
>>> d
csid(1: [1, 2, 3, 4, 5, 6])
```

If we just try to update the `csid` with the other dictionary, we would instead get this behavior:

```
>>> d = csid([[1, [1,2,3]]])
>>> d2 = csid([[1, [4,5,6]]])
>>> d.update(d2)
>>> d
csid(1: [4, 5, 6])
```

### Parameters

**other** (*dict*) – Other should be a dictionary structure where the values of each item in other are lists or dictionaries. Example: `other = {key: {sub-key1: value1, sub-key2: value2}}`

### `pop(key)`

Converts `key` to a `csiKeyString` or `csiKeyDecimal` if necessary, and then retrieves the entry in the `csid` instance and removes it using `dict.pop()`.

### Example

```
>>> from csid import csid
>>> d = csid([['a', 1], ['b', 2], ['c', 3]])
>>> d.pop('a')
1
>>> d
csid(csiKeyString('b'): 2, csiKeyString('c'): 3)
```

This will raise a `KeyError` if key is not in the `csid`:

```
>>> d.pop('a')
Traceback (most recent call last):
...
KeyError: csiKeyString('a')
```

### Returns

The value at `key`. The key, value pair is removed from the `csid`.

### `setdefault(key, default=None)`

Converts `key` to a `csiKeyString` or `csiKeyDecimal` if necessary, and then retrieves the entry in the `csid` instance. If `key` is not in the `csid`, insert `key` with a value of `default` and return `default`.

If *key* is in the *csid*, this simply retrieves its value:

```
>>> from csid import csid
>>> d = csid(['a', 1])
>>> d.setdefault('a')
1
```

If *key* is not in the *csid*, this will insert *key* and set the value to *default*:

```
>>> d.setdefault('b', 2)
2
>>> d
csid(csiKeyString('a'): 1, csiKeyString('b'): 2)
```

### Parameters

- **key** (*str*, *float*, *Decimal*) – The key for which to retrieve or set a value. *key* will be converted to a *csiKeyString* or *csiKeyDecimal* if necessary.
- **default** – The value which will be inserted into the *csid* for *key* if *key* is not already in the *csid*. Defaults to *None*.

### Returns

The value mapped to *key*, or *default* if *key* was not in the *csid*.

### update(*other*)

If *other* is not a *csid*, converts it to one, and then calls `update()` using the *csid* of *other*.

Example usage:

```
>>> from csid import csid
>>> d = csid(['a', 1])
>>> d.update(['b', 2])
>>> d
csid(csiKeyString('a'): 1, csiKeyString('b'): 2)
```

This function operates more quickly if *other* is already a *csid*, although the end result will be the same:

```
>>> d.update(csid(['c', 3]))
>>> d
csid(csiKeyString('a'): 1, csiKeyString('b'): 2, csiKeyString('c'): 3)
```

### Parameters

- **other** (*iterable*) – An iterable which is a *dict* type or can be used to construct a *csid*.

### \_\_contains\_\_(*key*)

Converts *key* to a *csiKeyString* or *csiKeyDecimal* if necessary, and then checks if *key* is in the *csid* instance.

Here's an example where we create a *csid* from a dictionary which uses an *int* as the key, and then we test if an *inpInt* with a value of the *int* is a key in the *csid*:

```
>>> from csid import csid
>>> from inpInt import inpInt
>>> from decimal import Decimal
```

(continues on next page)

(continued from previous page)

```
>>> d = csid({25: 'int', 'A1': 'str', 1.234: 'float'})
>>> inpInt(' 25') in d
True
>>> 'A1' in d
True
>>> Decimal('1.234') in d
True
```

If *key* is not in the *csid*, this will of course return False. The following example illustrates this, along with the fact that creating a *Decimal* directly from a *float* inherits the *float* inaccuracy:

```
>>> Decimal(1.234) in d
False
```

**Returns**

True if key in d else False.

**Return type**

bool

**\_\_delitem\_\_(key)**

Converts *key* to a *csiKeyString* or *csiKeyDecimal* if necessary, and then deletes the entry in the *csid* instance.

**Example**

```
>>> from csid import csid
>>> d = csid([[ 'a', 1], [ 'b', 2], [ 'c', 3]])
>>> del d['C']
>>> d
csid(csiKeyString('a'): 1, csiKeyString('b'): 2)
```

If *key* is not defined in the *csid*, this will raise a *KeyError* as expected:

```
>>> del d['D']
Traceback (most recent call last):
...
KeyError: csiKeyString('D')
```

**\_\_getitem\_\_(key)**

Converts *key* to a *csiKeyString* or *csiKeyDecimal* if necessary, and then retrieves the entry in the *csid* instance. Reference: `__getitem__()`.

**Example**

```
>>> from csid import csid
>>> d = csid([[ 'a', 1], [ 'b', 2], [ 'c', 3]])
>>> d['C']
3
```

If *key* is not defined in the *csid*, this will raise a `KeyError` as expected:

```
>>> d['D']
Traceback (most recent call last):
...
KeyError: csiKeyString('D')
```

---

**`__setitem__`**(*key*, *value*)

Converts *key* to a *csiKeyString* or *csiKeyDecimal* if necessary, and then creates `d[key] = value`.

---

### Example

```
>>> from csid import csid
>>> d = csid([[ 'a', 1], [ 'b', 2], [ 'c', 3]])
>>> d['d'] = 4
>>> d
csid(csiKeyString('a'): 1, csiKeyString('b'): 2, csiKeyString('c'): 3,
↪ csiKeyString('d'): 4)
```

---

**`__repr__`**()

Produces a string which can reproduce the *csid*.

---

### Example

```
>>> from csid import csid
>>> csid([[ 'a', 1], [ 'b', 2], [ 'c', 3]])
csid(csiKeyString('a'): 1, csiKeyString('b'): 2, csiKeyString('c'): 3)
```

---

**`__str__`**()

Creates a string for the *csid* where each (key, value) pair is on its own line.

---

### Example

```
>>> from csid import csid
>>> print(csid([[ 'a', 1], [ 'b', 2], [ 'c', 3], [4, 'd']]))
{
'a': 1,
'b': 2,
'c': 3,
4: 'd'
}
```

---

**`__weakref__`**

list of weak references to the object (if defined)

**class** `csid.csiKeyString`(*key*)

Bases: `str`

Creates a key derived from a string instance for *csid*.



**\_\_init\_\_**(*key*)Sets the *key* attr.**Example**

```
>>> from csid import csiKeyString
>>> csiKeyString(' Nodeset-1')
csiKeyString(' Nodeset-1')
```

*key* should be a string. Other types will raise an `AttributeError` when the other functions of this class are called. Example:

```
>>> a = csiKeyString(100)
>>> hash(a)
Traceback (most recent call last):
...
AttributeError: 'int' object has no attribute 'lower'
```

*key* is not automatically converted to a string for performance reasons. Therefore, if the user needs to use this class directly, they must first ensure that *key* is a string before instantiating it.

**Parameters**

**key** (*str*) – An instance of a string object to serve as the case- and space-insensitive key.

**key**

A string type object.

**Type***str***\_\_hash\_\_**()Hashes *rs1(key)*.

Strings which differ only in capitalization and spacing will hash to the same value. For example:

```
>>> from csid import csiKeyString
>>> hash(csiKeyString(' Nodeset-1')) == hash(csiKeyString('NODESET-1'))
True
```

Strings which have larger differences will not hash to the same value:

```
>>> hash(csiKeyString(' Nodeset-1')) == hash(csiKeyString('"NODESET-1'))
False
```

**Returns**The hashed value of *rs1(key)*.**Return type***int***\_\_eq\_\_**(*other*)Checks if *key* is equal to *other*.

*other* should be a *csiKeyString*. This function will not convert *other* to a *csiKeyString*, because in most cases this class should only be used through a *csid*, which will handle the conversion.

**Parameters**

**other** (*str*) – An instance of a string object to which *key* will be compared.

**Returns**

True if *rs1(key) == rs1(other)*, else False.

**Return type**

bool

---

**Examples**

Here's the typical usage:

```
>>> from csid import csKeyString
>>> csKeyString(' Nodeset-1') == csKeyString('Nodeset-1')
True
```

If *other* is not a *csKeyString*, this will convert *other* to a *csKeyString* before proceeding:

```
>>> csKeyString(' Nodeset-1') == 'Nodeset-1'
True
```

---

**\_\_str\_\_()**

Returns str(*key*).

---

**Example**

```
>>> from csid import csKeyString
>>> str(csKeyString(' Nodeset-1'))
' Nodeset-1'
```

---

**\_\_repr\_\_()**

Returns the string needed to recreate the object.

---

**Example**

```
>>> from csid import csKeyString
>>> csKeyString(' Nodeset-1')
csKeyString(' Nodeset-1')
```

---

**Returns**

str

---

**\_\_weakref\_\_**

list of weak references to the object (if defined)

**class** csid.**csKeyDecimal**(*key*)

Bases: *Decimal*

Creates a key derived from a float instance for *csid*. The key will be a precise *Decimal* implementation of str(float) as the float is written, as opposed to a *Decimal* of the evaluated float.

**static** `__new__(key)`

If *key* is a float, convert it to a string before creating the base `Decimal` instance.

**Parameters**

**key** (`float`, `Decimal`, `str`) – An object which will be converted to a `Decimal`-type.

**Return type**

*csiKeyDecimal*

`__init__(key)`

Does nothing by itself. `__new__()` does the necessary processing to the input.

If *key* is a float, it will be converted to a `Decimal`:

```
>>> from csid import csiKeyDecimal
>>> csiKeyDecimal(1.234)
csiKeyDecimal('1.234')
```

This also accepts `Decimal`-types as *key*:

```
>>> from inpDecimal import inpDecimal
>>> csiKeyDecimal(inpDecimal(' 1.234'))
csiKeyDecimal('1.234')
```

Strings are also acceptable keys:

```
>>> csiKeyDecimal(' 1.234')
csiKeyDecimal('1.234')
```

However, if the string does not evaluate to a `Decimal`, a `decimal.InvalidOperation` will be raised:

```
>>> csiKeyDecimal(' 1.234a')
Traceback (most recent call last):
...
decimal.InvalidOperation: [<class 'decimal.ConversionSyntax'>]
```

**Parameters**

**key** (`float`, `Decimal`, `str`) – Captures the input from `__new__`.

`__hash__()`

Hashes the *csiKeyDecimal* object.

**Example**

```
>>> from csid import csiKeyDecimal
>>> hash(csiKeyDecimal(1.234))
152185638608103802
```

The hash value will be identical using different input methods to *csiKeyDecimal*, provided the different input values will evaluate to the same `Decimal` value:

```
>>> from inpDecimal import inpDecimal
>>> hash(csiKeyDecimal(inpDecimal(' 1.234e0')))) == hash(csiKeyDecimal(' 12.
```

(continues on next page)

(continued from previous page)

```
↪34e-1'))  
True
```

**Returns**

The hashed value of the *csiKeyDecimal* object.

**Return type**

`int`

**`__eq__`(*other*)**

Converts *other* to a *csiKeyDecimal* (if necessary) and then returns `self.as_tuple() == other.as_tuple()`.

---

**Examples**

```
>>> from csid import csiKeyDecimal  
>>> csiKeyDecimal(' 12.34e-1') == 1.234  
True  
>>> csiKeyDecimal(' 12.34e-1') == 1.237  
False
```

**Parameters**

**other** (`float`, `Decimal`, `str`) – An instance of a `float`, `Decimal`, or `str` object to which the *csiKeyDecimal* will be compared.

**Returns**

True if `self.as_tuple() == other.as_tuple()`, else False.

**Return type**

`bool`

**`__str__`()**

Returns a string representation of the *csiKeyDecimal*.

---

**Example**

```
>>> from csid import csiKeyDecimal  
>>> str(csiKeyDecimal(' 12.34e-1'))  
'1.234'
```

**`__repr__`()**

Returns the string needed to recreate the object.

---

**Example**

```
>>> from csid import csiKeyDecimal  
>>> csiKeyDecimal(' 1.234')  
csiKeyDecimal('1.234')
```

**`__weakref__`**

list of weak references to the object (if defined)



## 17.1 Module Contents

`eval2.eval2(obj, t=None, ps=False, useDecimal=True)`

This function will convert a string into the appropriate data type.

This function is similar in concept to `eval()`, although it does not inherit from it. It produces different types than `eval()` to suit the needs of `inpRW`.

If `ps` is True, `eval2()` will return an `inpString`, `inpInt`, or `inpDecimal` as appropriate. If `ps` is False, `eval2()` will return `str`, `int`, and `float` (if `useDecimal = False`) or `Decimal` (if `useDecimal = True`).

If `t` is not set, `eval2()` will first try to convert `obj` to an integer type, and then a decimal type, and then finally a string type. If the underlying type of `obj` is known ahead of time, `t` can be set to this type to for better performance. However, if `t` is specified incorrectly, `obj` will be treated as a string. You should specify `t` when you are confident of the underlying type, as the function will run much faster.

Here are some example scenarios, first using `ps = True`:

```
>>> from eval2 import eval2
>>> eval2(' "Nodeset (1)"', t=str, ps=True)
inpString(' "Nodeset (1)"', False)
>>> eval2(' 100 ', t=int, ps=True)
inpInt(' 100 ')
>>> eval2(' 1.234E1 ', t=float, ps=True)
inpDecimal(' 1.234E1 ')
```

If we set `ps` and `useDecimal` to False, we get the built-in types:

```
>>> eval2(' "Nodeset (1)"', t=str, useDecimal=False)
' "Nodeset (1)"'
>>> eval2(' 100 ', t=int, useDecimal=False)
100
>>> eval2(' 1.234E1 ', t=float, useDecimal=False)
12.34
```

`useDecimal = True` only applies to floating point numbers, which will instead be evaluated to `Decimal`:

```
>>> eval2(' 1.234E1 ', t=float)
Decimal('12.34')
```

Not specifying `t` is slower, but `eval2()` will find the best type:

```
>>> eval2(' 1.23', ps=True)
inpDecimal(' 1.23')
```

If we specify *t* incorrectly, we will get a string type as the output:

```
>>> eval2(' 1.23', t=int, ps=True)
inpString(' 1.23', False)
```

### Parameters

- **obj** (*str*) – A string representation of the object to evaluate.
- **t** (*str or float or int*) – The expected type of *obj*. Pass in the type, not an object of the type.
- **ps** (*bool*) – Preserve Spacing. If True, *inpInt*, *inpDecimal*, and *inpString* objects will be created. Defaults to False.
- **useDecimal** (*bool*) – If True, *float*-like objects will instead be evaluated to *Decimal* objects. Defaults to True.

### Returns

*str*, *inpString*, *int*, *inpInt*, *Decimal*, or *inpDecimal*



## 18.1 Module Contents

The `elType` module provides the `elType` class, which stores information about an Abaqus element type.

**class** `elType.elType(name, numNodes, solvers="", description="")`

Bases: `object`

The `elType` class stores information about a particular Abaqus element type.

**\_\_init\_\_**(`name, numNodes, solvers, description=""`)

Initializes attributes of the `elType` class.

At the moment, `numNodes` is the only attribute used by `inpRW`. The other attributes are provided for the user's convenience.

If `numNodes` is an integer, it will be stored to `numNodes`. If `numNodes` is a set, it will be stored to `numNodesSet`.

### Parameters

- **name** (`str`) – The name of the element type.
- **numNodes** (`int` or `set`) – The number of nodes needed to define the element. This is either an integer or a set of integers if the element type can have a varying number of nodes.
- **solvers** (`str`) – A string indicating which Abaqus solvers support the element type. 'S' for Standard, 'E' for Explicit, and 'SE' for both.
- **description** (`str`) – The description string for the element type, taken from the Abaqus documentation.

---

### Examples

Here's the basic usage of the class:

```
>>> from elType import elType
>>> elType('AC1D2', 2, 'S', '    2-node acoustic link')
elType(name='AC1D2', numNodes=2, solvers='S', description='    2-node acoustic_
↪link')
```

If `numNodes` is a set, you need to use the `numNodesSet` attribute instead of `numNodes`:

```
>>> elType1 = elType('C3D15V', {16, 17, 18, 15}, 'S', '15 to 18-node_
↳triangular prism')
>>> l = list(elType1.numNodesSet)
>>> l.sort()
>>> l
[15, 16, 17, 18]
```

**name**

The element type name.

**Type**

str

**numNodesSet**

A set of all the numbers of nodes which can define the element.

**Type**

set

**numNodes**

The number of nodes needed to define the element.

**Type**

int

**description**

The description string for the element type, taken from the Abaqus documentation.

**Type**

str

**solvers**

A string indicating which Abaqus solvers support the element type. ‘S’ for Standard, ‘E’ for Explicit, and ‘SE’ for both.

**Type**

str

**\_\_getattr\_\_(name)**

Overrides the default attribute retrieval behavior for cases where *numNodes* is not defined.

This function raises an `AttributeError` guiding users towards the *numNodesSet* attribute if *numNodes* does not exist.

See `__getattr__()` for more information.

**Parameters**

**name** (str) – The attribute name.

**Raises**

`AttributeError` –

---

**Examples**

Here’s an example of trying to retrieve *numNodes* if it doesn’t exist:

```
>>> from elType import elType
>>> elType1 = elType('C3D15V', {16, 17, 18, 15}, 'S', '15 to 18-node_
↳triangular prism')
>>> elType1.numNodes
Traceback (most recent call last):
...
AttributeError: elType object has no attribute 'numNodes'. Use 'numNodesSet'
↳instead...
```

The error is slightly different for any other attribute:

```
>>> elType1.test
Traceback (most recent call last):
...
AttributeError: elType object has no attribute 'test'.
```

### `__repr__()`

Produces a repr of the *elType* instance.

If the instance includes *numNodesSet*, this will be reported as *numNodes*.

#### Returns

str

### Examples

Here's a basic example:

```
>>> from elType import elType
>>> elType('AC1D2', 2, 'S', '2-node acoustic link')
elType(name='AC1D2', numNodes=2, solvers='S', description='2-node acoustic_
↳link')
```

If the instance has *numNodesSet*, this will be listed as 'numNodes':

```
>>> elType('C3D15V', {16, 17, 18, 15}, 'S', '15 to 18-node triangular prism
↳')
elType(name='C3D15V', numNodes={16, 17, 18, 15}, solvers='S', description='
↳15 to 18-node triangular prism')
```

### `__str__()`

Produces a string of the *elType* instance.

Produces an identical string as `__repr__()`.

#### Returns

str

### Examples

```
>>> from elType import elType
>>> str(elType(name='C3D8R', numNodes=8, solvers='SE'))
"elType(name='C3D8R', numNodes=8, solvers='SE', description='')
```

---

**\_\_weakref\_\_**

list of weak references to the object (if defined)

## 19.1 Module Contents

**class** `inpDecimal.inpDecimal(inputStr)`

Bases: `Decimal`

An `inpDecimal` object behaves like a regular `Decimal`, except it has new attributes `_formatStr` and possibly `_formatExp`. These attributes are used to create exact string representations of the original object.

**static** `__new__(inputStr)`

This function processes the `inputStr` and handles some problematic strings.

It replaces ‘d’ and ‘D’ characters in `inputStr` prior to creating the base `Decimal` instance. It also raises an `DecimalInfError` if `inputStr` contains ‘INF’ (not case-sensitive). ‘INF’ will create a valid `Decimal` object, but there should be no cases where an infinite number is valid for an Abaqus input file. Thus, an exception is raised which triggers `inpRW` to handle the input as a string-like object instead of a `Decimal`.

The original `inputStr` will be passed to `__init__()`.

**Parameters**

**inputStr** (*str*) – The string to be processed. Should contain a floating-point like object.

**Raises**

`inpRWEErrors.DecimalInfError` –

`__init__(inputStr)`

Creates the `inpDecimal` instance.

This is like the `Decimal` class, except it also tracks the exact formatting of the original number string.

**Parameters**

**inputStr** (*str*) – The string representing the number which should be parsed.

---

**Examples**

```
>>> from inpDecimal import inpDecimal
>>> a = inpDecimal(' 3.14e0')
>>> str(a)
' 3.14e0'
```

Since `inpDecimal` inherits from `Decimal`, they support the same operations. Note that the resultant of most operations between two `inpDecimal` objects will be a `Decimal`, as shown in the following example:

```
>>> b = inpDecimal(' 2')
>>> a * b
Decimal('6.28')
```

Operations between an *inpDecimal* and a *float* will raise a *TypeError*, as expected:

```
>>> a * 2.0
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for *: 'inpDecimal' and 'float'
```

### **`_formatExp`**

Stores the exponent string if the original number string is in scientific notation. Defaults to None

### **`_formatStr`**

Stores the whitespace information of *inputStr*, with placeholder symbols (“%s”) where digits should be reinserted. Defaults to None

### **`_evalDecimal(inputStr)`**

Parses *inputStr* to recognize the numeric value and the original formatting.

This function will populate *\_formatExp* and *\_formatStr*, but it has no direct return. It will be called as part of *\_\_init\_\_()* and should not be called by itself.

Example usage:

```
>>> from inpDecimal import inpDecimal
>>> a = inpDecimal(' 1.234E-2')
```

This will set the following additional attributes, which track the exact formatting:

```
>>> a._formatStr
' %s.%s%s%sE%s '
>>> a._formatExp
'-2'
```

It also handles numbers which use “d” or “D” for the exponent notation:

```
>>> inpDecimal('5.678d+10')
inpDecimal('5.678d+10')
```

### **Parameters**

**`inputStr`** (*str*) – The string representing the number which should be parsed.

### **`_outstr()`**

Generates the output string. This should not be called by the user directly, as it is called automatically by *\_\_repr\_\_()* and *\_\_str\_\_()*.

Example usage:

```
>>> from inpDecimal import inpDecimal
>>> str(inpDecimal(' 1.234'))
' 1.234'
```

**Returns**

The string in the original formatting.

**Return type**

str

**\_\_repr\_\_()**

Creates a string representation which can be used to recreate the object.

**Example**

```
>>> from inpDecimal import inpDecimal
>>> inpDecimal(' -01.097 ')
inpDecimal(' -01.097 ')
```

**Return type**

str

**\_\_str\_\_()**

Calls `_outstr()` to create a string of object in the original formatting.

**Example**

```
>>> from inpDecimal import inpDecimal
>>> str(inpDecimal('\t-01.097 '))
'\t-01.097 '
```

**Return type**

str

**\_\_reduce\_\_()**

This function is required so the class can pickle/unpickle properly. This enables multiprocessing.

**Example**

```
>>> from inpDecimal import inpDecimal
>>> inpDecimal('5.678D+10').__reduce__()
(<class 'inpDecimal.inpDecimal'>, ('5.678D+10',))
```

**\_\_weakref\_\_**

list of weak references to the object (if defined)





## 20.1 Module Contents

**class** `inpInt.inpInt(inputStr)`

Bases: `int`

An `inpInt` object behaves like a regular `int`, except it has a new attribute `_formatStr`. This attribute is used to create exact string representations of the original object.

**static** `__new__(inputStr)`

Creates the `inpInt` instance.

`__init__(inputStr)`

`__init__(inputStr)`

Initializes the `inpInt` instance, represented by `inputStr`. Parses the integer value contained in `inputStr`, and also tracks any whitespace characters included in `inputStr`.

Example usage:

```
>>> from inpInt import inpInt
>>> num1 = inpInt(' 2 ')
>>> num1
inpInt(' 2 ')
>>> str(num1)
' 2 '
```

Since an `inpInt`, all of the typical `int` operations will work:

```
>>> num2 = inpInt(' 2 ')
>>> num1 + num2
4
```

Please note that most `int` operations will produce a `int`, not a `inpInt`:

```
>>> type(num1 + num2)
<class 'int'>
```

### Parameters

**`inputStr`** (*str*) – The string containing an integer which will be parsed, maintaining the original formatting.

**\_formatStr**

Stores the whitespace information of *inputStr*, with placeholder symbols (“%s”) where the integer value should be reinserted. Will include leading 0s, the positive sign, or the negative sign if it precedes leading 0s.

**\_outstr\_lead0()**

Generates the output string for cases which include a leading 0.

Example usage:

```
>>> from inpInt import inpInt
>>> str(inpInt(' -01'))
' -01'
```

**Returns**

The string in the original formatting.

**Return type**

str

**\_outstr()**

Generates the output string.

Example usage:

```
>>> from inpInt import inpInt
>>> str(inpInt(' 2 '))
' 2 '
```

**Returns**

The string in the original formatting.

**Return type**

str

**\_\_repr\_\_()**

Creates a string representation which can be used to recreate the object.

Example usage:

```
>>> from inpInt import inpInt
>>> inpInt(' 2 ')
inpInt(' 2 ')
```

**Return type**

str

**\_\_str\_\_()**

Calls *\_outstr()* to create a string of object in the original formatting.

Example usage:

```
>>> from inpInt import inpInt
>>> str(inpInt(' 2 '))
' 2 '
```

**Return type**

str

**\_\_getnewargs\_\_()**

This function is required so the class can pickle/unpickle properly. This enables multiprocessing.

---

**Example**

```
>>> from inpInt import inpInt
>>> inpInt(' 2 ').__getnewargs__()
(' 2 ',)
```

**Returns**

tuple



## 21.1 Module Contents

This module implements the `inpKeyword` class, which will parse and store information from an Abaqus keyword block. The module also contains a couple functions outside the class, but which are closely associated with the class.

`inpKeyword.createParamDictFromString(parameterString, ps=True, useDecimal=True)`

This function will create a `csid` from a string representing a keyword block's parameters.

Use `formatParameterOutput()` to write a string from the parameter dictionary.

### Parameters

- **parameterString** (*str*) – A string that contains the parameter information for a keyword block. This should include neither the keyword name nor the comma following the keyword name.
- **ps** (*bool*) – Short for Preserve Spaces. If True, `inpString`, `inpDecimal`, and `inpInt` classes will be used to preserve the exact spacing of the parent string when the items are parsed. Defaults to True.
- **useDecimal** (*bool*) – If True, all numbers that would evaluate to `float` will instead be `inpDecimal` or `Decimal`. Defaults to True.

### Returns

A case and space insensitive dictionary containing the parameter names (keys) and values.

### Return type

*csid*

---

### Examples

This shows the default behavior of the function.

```
>>> from inpKeyword import createParamDictFromString
>>> createParamDictFromString("change number=1000, pole")
csid(csiKeyString('change number'): inpInt('1000'), csiKeyString(' pole'): '')
```

If `ps` and `useDecimal` are both set to False, the items in `parameterString` will be parsed as `str`, `int`, and `float` instead of `inpString`, `inpInt`, and `inpDecimal`:

```
>>> createParamDictFromString("change number=1000, pole", ps=False,
↪ useDecimal=False)
csid(csiKeyString('change number'): 1000, csiKeyString(' pole'): '')
```

`inpKeyword.findElementNodeNum(t, lines)`

Returns the number of nodes needed to define an element in the given keyword block.

The element type is determined from `block.parameter['type']`. The number of nodes is then retrieved from `inpKeyword._elementTypeDictionary`, if it is defined, or it is estimated from the element data.

For variable node elements, this function looks up the valid number of nodes to define the element. Then, it keeps reading data lines until it finds a line which does not end with a comma. If the number of nodes is in the set of valid node numbers for the element type (`numNodesSet`), this function returns an integer. If the number of nodes is not in the set, an `ElementIncorrectNodeNumError` is raised.

For undefined element type labels, this function will read datalines until it finds one which does not end with a comma. It will return the number of nodes it thinks the element has. For accurate results with user or sub-structure elements, the user should add an entry to `_elementTypeDictionary` before calling `parse()`.

This function will only be called once per \*ELEMENT keyword block, so it assumes all elements in the keyword block are defined with the same number of nodes.

#### Returns

The number of nodes required for the element type. None: If the keyword block is not an ELEMENT keyword block.

#### Return type

int

#### Raises

`inpRWErrors.ElementIncorrectNodeNumError` –

---

### Examples

If the element type is defined in `~inpKeyword._elementTypeDictionary`, this function simply looks up the appropriate entry:

```
>>> from inpKeyword import findElementNodeNum
>>> lines = ['1, 22, 2, 10, 3, 45, 25, 33, 26',
...         '2, 5, 4, 15, 23, 28, 27, 38, 46']
>>> findElementNodeNum('C3D8R', lines)
8
```

If the element can have a variable number of nodes, this function will read from `lines` until it finds a line which ends without a comma. It then verifies that the number of nodes is found in `numNodesSet` of the element type entry:

```
>>> lines = ['101,101,102,103,104,105,106,107,108,',
...         '109,110,111,112,113,114,115,',
...         '201,202,203',
...         '**',
...         '102,103,102,117,106,105,116,108,122,',
...         '118,111,121,119,115,114,120,',
...         '202,204,205']
>>> findElementNodeNum('c3d15v', lines)
18
```

If the datalines are not formatted properly, this function will not be able to find the proper number of nodes and will raise an `ElementIncorrectNodeNumError`:

```
>>> lines = ['101,101,102,103,104,105,106,107,108,',
...          '109,110,111,112,113,114,115,',
...          '201,202,203,',
...          '**',
...          '102,103,102,117,106,105,116,108,122,',
...          '118,111,121,119,115,114,120,',
...          '202,204,205']
>>> findElementNodeNum('c3d15v', lines)
Traceback (most recent call last):
...
inpRWErrors.ElementIncorrectNodeNumError: ERROR! An element of type c3d15v must_
↳ have between 15 and 18 nodes.
```

For undefined element types (such as sub-structure or user elements), this function will keep reading datalines until one which does not end with a comma is encountered:

```
>>> lines = ['1, 1451,1452,1453,1454,1455,1456,1457,1483,1484,1485,1486,1487,',
...          '1488,1489,1596,1597,1629,1630,1652,1653,1681,1682,1704,1705,',
...          '1735,1736,1758,1759,2380,2381,2382,2383,2384,2385,2386,2412,',
...          '2413,2414,2415,2416,2417,2418,2525,2526,2558,2559,2581,2582,',
...          '2610,2611,2633,2634,2664,2665,2687,2688,7507,7508,7530,7531,',
...          '7551,7552,7574,7575,7973,7974,7996,7997,8017,8018,8040,8041']
>>> findElementNodeNum('Z1', lines)
WARNING! Element type 'Z1' is not well documented. It looks like this element type_
↳ needs 72 nodes to define the element.
    If this is incorrect, please specify the 'numNodes' attribute by running inp._
↳ elementTypeDictionary['Z1'] = elType(name='Z1', numNodes=NUM)
72
```

`inpKeyword.formatStringFromParamDict(parameters, joinPS=',', rmtrailing0=False)`

Creates a string from the Abaqus keyword *parameter* in valid keyword line formatting.

The dictionary should be a one-level structure. The output will be of the form key=item. If item is ',', then only key will appear in the output. key-item pairs are separated with joinPS, which defaults to ','.

#### Parameters

- **parameters** (*dict*) – The dictionary type whose contents should be formatted. Should be *parameter* or a portion of it.
- **joinPS** (*str*) – The string used to join parameter key-item pairs. Defaults to ','.
- **rmtrailing0** (*bool*) – If True, trailing 0s in number types will be omitted. Defaults to False.

#### Returns

str

#### Examples

This shows the creation of a parameter dictionary, modifying one of the values in the dictionary, and creating a string from the dictionary:

```
>>> from inpKeyword import *
>>> d = createParamDictFromString('INC=2000, NLGEOM, UNSYMM=YES')
```

(continues on next page)

(continued from previous page)

```
>>> d['inc'] = int(d['inc'] / 2)
>>> formatStringFromParamDict(d)
'INC=1000, NLGEOM, UNSYMM=YES'
```

`inpKeyword._insertComments(data, comments)`

This function will insert each item in *comments* to the appropriate location in *data*.

*data* should be a list of strings from the data from an *inpKeyword*, while *comments* should be *inpKeyword.comments*.

The user should not call this function in most circumstances. The `_formatDataOutput()` and `_formatDataLabelDict()` functions are meant to call this function.

#### Parameters

- **data** (*list*) – A *list* of *strings*. This list should be produced from the `formatData` function inside `_formatDataOutput()` or `_formatDataLabelDict()`.
- **comments** (*list*) – A *list* of *lists* of the form `[[int, str], ...]`. Should be *comments*.

#### Returns

A *list* of *strings* corresponding to the *data* and *comments* properly ordered.

#### Return type

*list*

---

#### Examples

We'll create an *inpKeyword* which contains a comment line:

```
>>> from inpKeyword import inpKeyword
>>> block = inpKeyword(' '*ELEMENT, TYPE=C3D4
... 1, 1, 2, 3, 4
... **2, 5, 6, 7, 8
... 3, 9, 10, 11, 12')
```

If we print *data* and *comments*, we can verify the data and comment lines are stored separately:

```
>>> print(block.data)
{
1: 1, 1, 2, 3, 4
3: 3, 9, 10, 11, 12
}
>>> print(block.comments)
[[1, '**2, 5, 6, 7, 8']]
```

Finally, calling `formatData()` will automatically call the appropriate sub-function for formatting the datalines and comments. The sub-function will automatically call `_insertComments()` to place the comment strings to their original location:

```
>>> block.formatData()
['\n1, 1, 2, 3, 4', '\n**2, 5, 6, 7, 8', '\n3, 9, 10, 11, 12']
```



Since the position of the comment lines is stored absolutely, if we delete a data line, the comment will likely not end up in the same relative position:

```
>>> del block.data[1]
>>> block.formatData()
['\n3, 9, 10, 11, 12', '\n**2, 5, 6, 7, 8']
```

```
class inpKeyword.inpKeyword(inputString="", name="", parameter=None, data=None, path="",
                             suboptions=None, comments=None, createSuboptions=True, **inpRWargs)
```

This class creates a blank *inpKeyword* object.

The *inpKeyword* object is almost identical in the structure to that created by the *inpParser* module, except data containers are mostly mutable instead of tuples.

Using print on an instance of this object will create a string of the *inpKeyword* in valid Abaqus formatting. This could potentially print a huge amount of data, so the user should have some idea what the *inpKeyword* instance holds before printing it. This applies to any operation that will trigger `__str__()`.

This will print all information corresponding to the *inpKeyword* block, but not any information from *suboptions*. This will include *data* that was read and/or will be written to a sub-input file as specified by the INPUT parameter.

```
__init__(inputString="", name="", parameter=None, data=None, path="", suboptions=None,
          comments=None, createSuboptions=True, **inpRWargs)
```

Initializes the *inpKeyword*.

There are three methods to initialize *inpKeyword*. The first is to specify no input arguments to create a blank *inpKeyword*. The second is to specify any or all of *name*, *parameter*, *data*, *path*, *suboptions*, and *comments* directly; if using this option, each of the items will need to be formatted properly prior to insertion. The third option is to specify *inputString*, which is a string consisting of the entirety of the Abaqus keyword block in proper Abaqus input file syntax. If *inputString* is specified, the `__init__` process will handle parsing *inputString* and populating the appropriate attributes, thus sparing the user this task. The user can also pass additional arguments to this function to control certain formatting options. See the *inpRWargs* entry in the Args section for more details.

The most important attributes of *inpKeyword* are the following:

*name*, *parameter*, *data*, *comments*, *path*, *suboptions*

Users might need to be aware of these attributes (especially if creating keywords from scratch), but they should not often need to access them directly:

*\_inpItemsToUpdate*, *\_data*, *\_subdata*, *\_dataParsed*

The most important user functions are the following:

*parseKWData()*, *formatData()*, *formatKeywordLine()*, *formatParameterOutput()*,  
*writeKWandSuboptions()*, *\_setMiscInpKeywordAttrs()*

### Parameters

- **inputString** (*str*) – The string corresponding to the entirety of the keyword block. This must be one string, not a sequence of strings. Defaults to “”.
- **name** (*str*) – The keyword name. Defaults to “”.
- **parameter** (*csid*) – Stores the parameters and their values. Defaults to an empty *csid*.
- **data** (*list*) – A list of lists to hold dataline information. Each sublist corresponds to one dataline. Defaults to [].

- **path** (*str*) – Stores the path to access the keyword block in *keywords*. Defaults to “”.
- **suboptions** (*list*) – Stores the sub blocks to the *inpKeyword* object if *organize* = True. Defaults to *inpKeywordSequence*.
- **comments** (*list*) – Stores the comments of the *inpKeyword*. Defaults to [].
- **createSuboptions** (*bool*) – If True, will set the *suboptions* attribute to a blank *inpKeywordSequence* instance. If False, *suboptions* will be None. Defaults to True.
- **inpRWargs** (*dict*) – Should contain attributes from *inpRW* that are needed for proper formatting and organization of the parsed data. *parse()* will pass the appropriate values to this function automatically. *inpKeyword* can be used without any of these attributes specified, in which case default values that allow a standalone *inpKeyword* will be used. To be consistent with the rest of the parsed keyword blocks, you should pass the *inpKeywordArgs* dictionary to this function as follows

```
inpKeyword(arg1, arg2, argn, **inp.inpKeywordArgs)
```

## Examples

Here’s an example of creating an *inpKeyword* by specifying the *inputString* argument:

```
>>> from inpKeyword import inpKeyword
>>> blocka = inpKeyword(' '*Node, nset=Nset-1
... 1, 0.0, 0.0, 0.0
... **Comment
... 2, 1.0, 0.0, 0.0')
>>> blocka
inpKeyword(name='Node', parameter=csid(csiKeyString(' nset'): inpString('Nset-1
↳ ', False)), data=csid(inpInt('1'): Node(label=inpInt('1'), data=[inpInt('1'),
↳ inpDecimal(' 0.0'), inpDecimal(' 0.0'), inpDecimal(' 0.0')], elements=[]),
↳ inpInt('2'): Node(label=inpInt('2'), data=[inpInt('2'), inpDecimal(' 1.0'),
↳ inpDecimal(' 0.0'), inpDecimal(' 0.0')], elements=[])), path='', comments=[[1,
↳ '**Comment']], suboptions=inpKeywordSequence(num child keywords: 0, num
↳ descendant keywords: 0, path: .suboptions))
>>> str(blocka)
'\n*Node, nset=Nset-1\n1, 0.0, 0.0, 0.0\n**Comment\n2, 1.0, 0.0, 0.0'
```

We can create the equivalent block if we forgo the *inputString* argument and instead specify the *name*, *parameter*, *data*, and *comments* arguments, although this will be more work for the user in many cases. Example:

```
>>> from inpKeyword import createParamDictFromString
>>> from mesh import *
>>> name = 'Node'
>>> parameter = createParamDictFromString(' nset=Nset-1')
>>> node1 = Node('1, 0.0, 0.0, 0.0')
>>> node2 = Node('2, 1.0, 0.0, 0.0')
>>> data = Mesh([i.label, i] for i in [node1, node2])
>>> comments = [[1, '**Comment']]
>>> blockb = inpKeyword(name=name, parameter=parameter, data=data,
↳ comments=comments)
>>> str(blocka) == str(blockb)
True
```

**preserveSpacing**

See *inpRW.preserveSpacing*. Should be passed from the *inpRW* instance via *inpRWargs*. Defaults to True

**useDecimal**

See *inpRW.useDecimal*. Should be passed from the *inpRW* instance via *inpRWargs*. Defaults to True

**\_ParamInInp**

See *inpRW.\_ParamInInp*. Should be passed from the *inpRW* instance via *inpRWargs*. Defaults to False

**fastElementParsing**

See *inpRW.fastElementParsing*. Should be passed from the *inpRW* instance via *inpRWargs*. Defaults to True

**\_joinPS**

See *inpRW.\_joinPS*. Should be passed from the *inpRW* instance via *inpRWargs*. Defaults to ‘,’

**parseSubFiles**

See *inpRW.parseSubFiles*. Should be passed from the *inpRW* instance via *inpRWargs*. Defaults to False

**inputFolder**

See *inpRW.inputFolder*. Should be passed from the *inpRW* instance via *inpRWargs*. Defaults to ‘’

**\_nl**

See *inpRW.\_nl*. Should be passed from the *inpRW* instance via *inpRWargs*. Defaults to ‘n’

**rmtrailing0**

See *inpRW.rmtrailing0*. Should be passed from the *inpRW* instance via *inpRWargs*. Defaults to False

**\_addSpace**

See *inpRW.\_addSpace*. Should be passed from the *inpRW* instance via *inpRWargs*. Defaults to ‘’

**\_debug**

See *inpRW.\_debug*. Should be passed from the *inpRW* instance via *inpRWargs*. Defaults to False

**delayParsingDataKws**

See *inpRW.delayParsingDataKws*. Should be passed from the *inpRW* instance via *inpRWargs*. Defaults to set()

**\_subinps**

Contains each sub *inpRW* instance associated for this block. Only populated for \*INCLUDE and \*MANIFEST.

**Type**

list

**\_firstItem**

See *inpRW.\_firstItem*. Defaults to False

**\_leadstr**

Stores any whitespace characters that occur on a keyword line prior to the \* symbol. Defaults to None

**Type**

str

**\_baseStep**

Stores the base step for \*STEP keyword blocks.

**Type***inpKeyword***\_namedRefs**

See *inpRW.namedRefs*. Defaults to `csid()`

**\_inpItemsToUpdate**

Contains the items from the *inpKeyword* block that need to be set in the *inpRW* instance. Defaults to `{'namedRefs': self._namedRefs}`

**Type***dict***pd**

See *inpRW.pd*. Defaults to `None`

**\_data**

A sub-instance of *inpKeyword* that contains the *data* and *comments* of the keyword block that are in the main input file if the keyword block is set to read data from a sub-file using the INPUT parameter. Does not exist by default.

**Type***inpKeyword***\_subdata**

A sub-instance of *inpKeyword* that contains the *data* and *comments* of the keyword block that are in the sub input file if the keyword block is set to read data from a sub-file using the INPUT parameter. Does not exist by default.

**Type***inpKeyword***\_nd**

A *Mesh* instance which will hold node labels and the elements connected to them. Only exists for \*ELEMENT keyword blocks.

**Type***Mesh***inputString**

The unparsed string representing the entirety of the keyword block. Will be the value of the *inputString* parameter for `__init__()`. Defaults to ""

**Type***str***name**

The name of the keyword block (i.e. "ELEMENT" for a \*ELEMENT keyword block). Defaults to ""

**Type***str***parameter**

Stores the keyword parameters and their values. Defaults to `csid()`

**Type***csid*

**data**

Stores the parsed data. Each item in *data* will be a list containing the parsed entries from one dataline. If the keyword block is \*NODE or \*ELEMENT, *data* will instead be a *Mesh* instance. If the keyword block specifies data in a sub-input file via the INPUT parameter, *data* will first list the data from the main input file, and then the data from the sub-input file. Defaults to []

**Type**

list or *Mesh*

**\_dataParsed**

Indicates if the data for the keyword block has been parsed. Will be set to True by *parseKWData()* if the keyword's data was parsed. Defaults to False

**Type**

bool

**path**

Stores a string representation of the path to the *inpKeyword* object through *keywords*. Defaults to ''

**Type**

str

**suboptions**

Stores the sub blocks to the *inpKeyword* object if *organize* = True. The suboptions for each block will be set during the block organization loop of *parse()*. Defaults to *inpKeywordSequence*

**Type**

*inpKeywordSequence*

**comments**

Stores the commented lines with the keyword block (i.e. any line that starts with "\*\*\*" following the keyword line, but before the next keyword line. Each item in comments will be of the form [ind, line] where ind is the index of the line in *data* and line is the string of the comment line. If the keyword block specifies data in a sub-input file via the INPUT parameter, *comments* will first list the comments from the main input file, and then the comments from the sub-input file. Defaults to []

**Type**

list

**lines**

Stores the input strings for the keyword block; these will have been split on new line characters. This is normally deleted when the block is parsed, but it will include the unparsed strings representing the datalines if the data was not parsed.

**Type**

list

**formatData(includeSubData=True)**

This function maps the actual formatting function used by a given keyword block to a consistent name (formatData).

If the *inputString* or *name* parameters are specified, formatData will actually be an attribute mapping to the appropriate function for formatting the data of the keyword block. This function exists solely to create the mapping to the real function and call the real function. This function will not be used in most cases.

**Parameters**

**includeSubData** (*bool*) – If True, data from *\_subdata* will also be included in the output. Defaults to True.

**Returns**

A list with a string for each line in *data*.

**Return type**

list

---

**Examples**

This example shows creating a keyword block by populating the individual fields:

```
>>> from inpKeyword import inpKeyword, createParamDictFromString
>>> name = 'Nset'
>>> parameters = createParamDictFromString('nset= "Nset 2"')
>>> data = [[1,2,3,4,5]]
>>> block = inpKeyword(name=name, parameter=parameters, data=data)
>>> block.formatData()
['\n1,2,3,4,5']
```

Any method which populates the *data* attribute will automatically set *formatData()* to the appropriate sub-function. We can see this for the previous keyword block:

```
>>> block.formatData.__name__
'_formatDataOutput'
```

*formatData* is mapped to *\_formatDataLabelDict()* for \*NODE and \*ELEMENT keyword blocks:

```
>>> block2 = inpKeyword('*ELEMENT, elset=test, type=C3D8R\n1, 1, 2, 3, 4, 5, 6,\n7, 8')
>>> block2.formatData.__name__
'_formatDataLabelDict'
```

The only case where this is not set automatically is if we create a blank *inpKeyword* instance and then populate it. In this case, *formatData()* will not be remapped until it has been called once:

```
>>> block3 = inpKeyword()
>>> block3name = 'ELSET'
>>> block3parameter = createParamDictFromString(' elset=test_elset')
>>> block3data = [[1, 2, 3, 4, 5, 6, 7, 8]]
>>> block3.name = block3name
>>> block3.parameter = block3parameter
>>> block3.data = block3data
>>> block3.formatData.__name__
'formatData'
```

Now calling *formatData()* picks the correct sub-function based on the element name and remaps the *formatData* name from this function to the sub-function:

```
>>> block3.formatData()
['\n1,2,3,4,5,6,7,8']
>>> block3.formatData.__name__
'_formatDataOutput'
```

---

**formatKeywordLine()**

Creates a printable string for the keyword line(s) of the *inpKeyword* instance.

Keyword lines include the keyword name and the parameters, but not the datalines. If the *inpKeyword* instance originally had multiple keyword lines, this function will also write out multiple keyword lines.

#### Returns

A string representing the keyword line(s).

#### Return type

str

---

#### Examples

First we create a sample keyword block, and then call the function:

```
>>> from inpKeyword import inpKeyword
>>> block = inpKeyword(' '*Nset, nset= "Nset 2"
... 1, 2, 3, 4, 5'')
>>> block.formatKeywordLine()
'\n*Nset, nset= "Nset 2"'
```

This will also handle cases where the keywordline is split across multiple lines:

```
>>> block = inpKeyword(' '*Element, type=C3D8,
... elset= Elset-1
... 1, 1, 2, 3, 4, 5, 6, 7, 8'')
>>> block.formatKeywordLine()
'\n*Element, type=C3D8,\nelset= Elset-1'
```

If *name* is not set, this function simply returns a blank string:

```
>>> block = inpKeyword()
>>> block.formatKeywordLine()
''
```

---

#### formatParameterOutput()

Turns the parameter dictionary into a printable string.

#### Returns

A string representation of the parameter dictionary.

#### Return type

str

---

#### Examples

You can use this function if you need to create a string just from the parameters. The parameters will be written in their original order:

```
>>> from inpKeyword import inpKeyword
>>> block = inpKeyword(' '*STEP, INC=2000, NLGEOM, UNSYMM=YES
... STEP 3 - EARTHQUAKE'')
>>> block.formatParameterOutput()
' INC=2000, NLGEOM, UNSYMM=YES'
```

**parseKWData**(*parseDelayed=False*)

This function calls `_parseData()` to parse keyword block data, and adds the results of that function call to `data`. It is called during `__init__()` if the `inputString` parameter is not ''.

This function will not parse the keyword block data if `name` is in `delayParsingDataKws` and `parseDelayed = False` (the default). This will allow the user to avoid costly parsing operations if they only need access to the organization of the keywords. If the data is not parsed, it will be stored as a list of strings in the `lines` attribute, and this list will be included when a string of the entire keyword block is produced.

---

**Note:** You should only place keyword block names in `delayParsingDataKws` if you know for certain you will not need to access the data in those keyword blocks. `inpRW` currently does not include a mechanism to automatically parse the data of keyword blocks after the `inpKeyword` has been initialized. This will likely be enhanced in the future.

---

If `name` is in `_dataKws` and the keyword includes the INPUT parameter, `_parseData()` will delete the `data` attribute and create `_data` and `_subdata`, which are sub-instances of `inpKeyword`. The data and comments from the main input file will be stored in `_data`. If `parseSubFiles = True`, this function will also read the data and comments from the sub-file (whose name is specified by the value of the INPUT parameter) and store them to `_subdata`. The user should rarely need to directly access `_data` and `_subdata`. If `data` or `comments` are not set, `__getattr__()` will retrieve the information from `_data` and `_subdata` automatically.

**Parameters**

**parseDelayed** (*bool*) – If True, will parse data for any blocks which were set to parse on a later loop. This is controlled by `parse()` and `delayParsingDataKws`. Defaults to False.

**Returns**

None if `data` should not be populated at this time, else no return.

**Return type**

`bool`

**Examples**

In the most basic case, we parse a simple keyword block, which calls this function automatically:

```
>>> from inpKeyword import inpKeyword
>>> block1 = inpKeyword(r'''*NSET,NSET=MIDPLANE
... 204,303,402,214,315,416
... 1902,2003,1916,2015
... *''')
>>> block1.data
[[inpInt('204'), inpInt('303'), inpInt('402'), inpInt('214'), inpInt('315'),
↳ inpInt('416')], [inpInt('1902'), inpInt('2003'), inpInt('1916'), inpInt('2015
↳ ')]]
```

If a keyword block has some data and/or comments in the main input file, but some data in a separate file specified by the INPUT parameter, the information from each file will be stored in separate `_data` and `_subdata` attributes (these keyword names are store in `config._dataKws`). The user can still simply use `data` to access the combined information; the information from the main input file will always be reported first. When the input file is written, any information in `_subdata` will be written to the file specified by the INPUT parameter. Example:



```

>>> from config import _slash as sl
>>> block2 = inpKeyword(f''*NODE,NSET=HANDLE,INPUT=sample_input_files{sl}
↳ inpKeyword{sl}tennis_rig1.inp
... **
... **'', parseSubFiles=True)
>>> print(block2)

*NODE,NSET=HANDLE,INPUT=sample_input_files...inpKeyword...tennis_rig1.inp
**
**
50001,      0.54,      -8.425,      0.25
50002,      0.54,     -9.80625,      0.25
50003,      0.54,    -11.1875,      0.25

```

We can verify the information is stored in separate containers:

```

>>> print(block2._data)

**
**
>>> print(block2._subdata.data)
{
  50001:    50001,      0.54,      -8.425,      0.25
  50002:    50002,      0.54,     -9.80625,      0.25
  50003:    50003,      0.54,    -11.1875,      0.25
}

```

Of course, if the INPUT parameter is not part of the keyword block, `_data` and `_subdata` won't be created:

```

>>> block3 = inpKeyword(''*NODE
... ** Bottom curve.
... 104, -2.700,-6.625,0.
... 109, 0.000,-8.500,0.
... 114, 2.700,-6.625,0.'')
>>> hasattr(block3, '_data')
False

```

If you won't need to access the data of a keyword type and wish to avoid parsing all blocks with that name, you can include that name in `delayParsingDataKws`. You would typically set this at the `inpRW`, but it's shown here for illustration. This will save some processing time, especially for keyword blocks with a lot of data.

```

>>> block4 = inpKeyword(''*NODE
... ** Bottom curve.
... 104, -2.700,-6.625,0.
... 109, 0.000,-8.500,0.
... 114, 2.700,-6.625,0.'', delayParsingDataKws={'node'})
>>> block4.data
csid()

```

The data is still stored as a list of unparsed strings in the `lines` attribute, so it will still be included when a string of the `inpKeyword` block is produced.

```
>>> print(block4)

*NODE
** Bottom curve.
104, -2.700,-6.625,0.
109, 0.000,-8.500,0.
114, 2.700,-6.625,0.
```

---

**Todo:** Replace return None with raise a custom exception type.

---

### `printKWandSuboptions(includeSubData=False)`

This function will print the list of strings generated by `writeKWandSuboptions()`.

#### Parameters

**includeSubData** (*bool*) – If True, will include *data* read from a sub file in the output string list. Defaults to False.

---

#### Examples

We'll first parse an input file:

```
>>> import inpRW
>>> from config import _slash as sl
>>> inp = inpRW.inpRW(f'sample_input_files{sl}inpKeyword{sl}mcooot3vlp.inp')
>>> inp.parse()

Splitting string of input file from ... into strings representing each keyword_
↳block.

Parsing 36 keyword blocks using 1 cores.
...
Updating keyword blocks' paths
Searching for *STEP locations

Finished parsing mcooot3vlp.inp
time to parse = ...
```

Next, we'll find the *\*STEP* keyword containing the *\*TEMPERATURE* block and call `printKWandSuboptions()` on it:

```
>>> for tempblock in inp.kwg['temperature']: # Each kwg entry is a set, so we_
↳must iterate through it to select the only TEMPERATURE block
...     break
>>> step = inp.getParentBlock(tempblock, parentKWName='step')
>>> step.printKWandSuboptions()
*STEP,INC=10
STEP 3 - REST OF NONLINEAR
*STATIC,DIRECT
1.,10.
*BOUNDARY
```

(continues on next page)

(continued from previous page)

```

7,3,,-.005
5,3,,-.005
6,3,,-.005
8,3,,-.005
*TEMPERATURE,INPUT=mcooot3vlp_temp.inp
*END STEP

```

If we activate the *includeSubData* parameter, the contents of sub-input files will be shown in the output:

```

>>> step.printKWandSuboptions(includeSubData=True)
*STEP,INC=10
STEP 3 - REST OF NONLINEAR
*STATIC,DIRECT
1.,10.
*BOUNDARY
7,3,,-.005
5,3,,-.005
6,3,,-.005
8,3,,-.005
*TEMPERATURE,INPUT=mcooot3vlp_temp.inp
1,20.
2,20.
3,20.
4,20.
5,20.
6,20.
7,20.
8,20.

*END STEP

```

### **writeKWandSuboptions**(*firstItem=False, includeSubData=False*)

This function will create a list of strings representing each line of the *inpKeyword*, and all blocks in *suboptions*, except for *suboptions* of \*INCLUDE or \*MANIFEST blocks. If *includeSubData* is True, the data read from sub files will also be included.

The first character of the output will be a new line character, unless *firstItem* = True. This prevents adding an extra new line character to the start of the text block or input file.

#### **Parameters**

- **firstItem** (*bool*) – If True, a new line character will not be written to the start of the output string list. Defaults to False.
- **includeSubData** (*bool*) – If True, will include *data* read from a sub file in the output string list. Defaults to False.

#### **Returns**

A list of strings.

#### **Return type**

*list*

#### **Examples**

We'll first parse an input file:

```
>>> import inpRW
>>> from config import _slash as sl
>>> inp = inpRW.inpRW(f'sample_input_files{sl}inpKeyword{sl}mcooot3vlp.inp')
>>> inp.parse()

Splitting string of input file from ... into strings representing each keyword_
↳block.

Parsing 36 keyword blocks using 1 cores.
...
Updating keyword blocks' paths
Searching for *STEP locations

Finished parsing mcooot3vlp.inp
time to parse = ...
```

Next, we'll find the \*STEP keyword containing the \*TEMPERATURE block and call `printKWandSuboptions()` on it:

```
>>> for tempblock in inp.kwg['temperature']: # Each kwg entry is a set, so we_
↳must iterate through it to select the only TEMPERATURE block
...     break
>>> step = inp.getParentBlock(tempblock, parentKWName='step')
>>> step.writeKWandSuboptions()
['\n*STEP,INC=10', '\nSTEP 3 - REST OF NONLINEAR', '\n*STATIC,DIRECT', '\n1.,10.
↳', '\n*BOUNDARY', '\n7,3,-.005', '\n5,3,-.005', '\n6,3,-.005', '\n8,3,-.
↳005', '\n*TEMPERATURE,INPUT=mcooot3vlp_temp.inp', '\n*END STEP']
```

`firstItem = True` will prevent a newline character from being added to the first output item:

```
>>> step.writeKWandSuboptions(firstItem=True)
['*STEP,INC=10', '\nSTEP 3 - REST OF NONLINEAR', '\n*STATIC,DIRECT', '\n1.,10.',
↳ '\n*BOUNDARY', '\n7,3,-.005', '\n5,3,-.005', '\n6,3,-.005', '\n8,3,-.005
↳', '\n*TEMPERATURE,INPUT=mcooot3vlp_temp.inp', '\n*END STEP']
```

If we activate the `includeSubData` parameter, the contents of sub-input files will be shown in the output:

```
>>> step.writeKWandSuboptions(includeSubData=True)
['\n*STEP,INC=10', '\nSTEP 3 - REST OF NONLINEAR', '\n*STATIC,DIRECT', '\n1.,10.
↳', '\n*BOUNDARY', '\n7,3,-.005', '\n5,3,-.005', '\n6,3,-.005', '\n8,3,-.
↳005', '\n*TEMPERATURE,INPUT=mcooot3vlp_temp.inp', '\n1,20.', '\n2,20.', '\n3,
↳20.', '\n4,20.', '\n5,20.', '\n6,20.', '\n7,20.', '\n8,20.', '\n', '\n*END_
↳STEP']
```

`_cloneKW(attributes=None)`

`_cloneKW(attributes=None)`

Creates a new keyword object which is an identical copy of self.

#### Parameters

**attributes** (*list*) – A list indicating the attributes of self to copy to the new keyword

object. If None, all attributes will be copied. Valid attributes are 'name', 'parameter', 'data', 'path', 'suboptions', and 'comments'. Defaults to None.

#### Returns

inpKeyword

---

**Todo:** This function needs to be reworked. There are more robust ways to implement it. Use with caution.

---

#### `_formatDataOutput(includeSubData=True)`

Creates a list of printable strings for the datalines of the `inpKeyword` instance.

New line characters will be prepended to the string for each line, unless the `inpKeyword` instance has no keyword line and this is the first block in the input file.

#### Parameters

**includeSubData** (*bool*) – If True, the `data` and `comments` in the `_subdata` (i.e. data lines read from a sub file) will be included in the output. They will be included after the `data` and `comments` from the main input file. Defaults to True.

#### Returns

A list of strings, with each item corresponding to a dataline.

#### Return type

list

---

#### Examples

In the most basic case, we parse a simple keyword block and then call this function:

```
>>> from inpKeyword import inpKeyword
>>> block1 = inpKeyword(r'''*NSET,NSET=MIDPLANE
... 204,303,402,214,315,416
... 1902,2003,1916,2015
... **''')
>>> block1._formatDataOutput()
['\n204,303,402,214,315,416', '\n1902,2003,1916,2015', '\n**']
```

This function will also format data and comments from sub files if `includeSubData = True` (the default):

```
>>> from config import _slash as sl
>>> block2 = inpKeyword(fr'''*TEMPERATURE,INPUT=sample_input_files{sl}inpKeyword
↳ {sl}mcooot3vlp_temp.inp
... **'', parseSubFiles=True)
>>> block2._formatDataOutput()
['\n**', '\n1,20.', '\n2,20.', '\n3,20.', '\n4,20.', '\n5,20.', '\n6,20.', '\n7,
↳ 20.', '\n8,20.', '\n']
```

If we call it with `includeSubData = False`, this will return just the information from the main block:

```
>>> block2._formatDataOutput(includeSubData=False)
['\n**']
```

---

#### `_formatDataLabelDict(includeSubData=True)`

`_formatDataLabelNodeDict(includeSubData=True)`

This function will create strings for the datalines for \*NODE and \*ELEMENT keyword blocks, which have their *data* stored in a dictionary-like construct.

New line characters will be prepended to the string for each line, unless the *inpKeyword* instance has no keyword line and this is the first block in the input file.

#### Parameters

**includeSubData** (*bool*) – If True, the *data* and *comments* in the *\_subdata* (i.e. data lines read from a sub file) will be included in the output. They will be included after the *data* and *comments* from the main input file. Defaults to True.

#### Returns

A list of strings, with each item corresponding to a dataline.

#### Return type

list

### Examples

In the most basic case, we parse a simple keyword block and then call this function:

```
>>> from inpKeyword import inpKeyword
>>> block1 = inpKeyword('*NODE
... ** Bottom curve.
... 104, -2.700,-6.625,0.
... 109, 0.000,-8.500,0.
... 114, 2.700,-6.625,0.')
>>> block1._formatDataLabelDict()
['\n** Bottom curve.', '\n 104, -2.700,-6.625,0.', '\n 109, 0.000,-8.500,0.',
↪ '\n 114, 2.700,-6.625,0.']
```

This function will also format data and comments from sub files if *includeSubData* = True (the default):

```
>>> from config import _slash as sl
>>> block2 = inpKeyword(fr'*NODE,NSET=HANDLE,INPUT=sample_input_files{sl}
↪ inpKeyword{sl}tennis_rig1.inp
... **
... **'', parseSubFiles=True)
>>> block2._formatDataLabelDict()
['\n**', '\n**', '\n 50001, 0.54, -8.425, 0.25', '\n ↪
↪ 50002, 0.54, -9.80625, 0.25', '\n 50003, 0.54, -
↪ 11.1875, 0.25']
```

If we call it with *includeSubData* = False, this will return just the information from the main block:

```
>>> block2._formatDataLabelDict(includeSubData=False)
['\n**', '\n**']
```

### handleComment(*line*, *ind*)

Appends a list of [*int*, *str*] to *comments* of the keyword block. The *int* represents the position of the comment in the data, and the *str* is the unparsed string of the commented line.

#### Parameters

- **line** (*str*) – The comment data line as a string.
- **ind** (*int*) – The integer representing the position of *line* in *data*.

**`_handleCommentSub(line, ind)`**

Appends a list of [`int`, `str`] to `comments` of the `_data` keyword block. The `int` represents the position of the comment in the data, and the `str` is the unparsed

**Parameters**

- **line** (`str`) – The comment data line as a string.
- **ind** (`int`) – The integer representing the position of `line` in `data`.

**`_parseData()`**

This function maps the actual parsing function used by a given keyword block to a consistent name (`_parseData`).

It will be overridden by `_parseKWLine()`. In case `_parseKWLine()` is not called (for example, the attributes of the keyword block are populated manually instead of parsed from a string), this function will overwrite itself with the call to the proper parsing function based on `name`.

**`_parseDataElement()`**

Parses the data in \*ELEMENT blocks.

This function will parse the data in the \*ELEMENT block. It will also add some information to `_nd`. Comments are left as strings and added to the appropriate `comments` section.

This function will read a dataline and check if the number of nodes in that dataline matches the element type's specification. If not, the next dataline will be read until the appropriate number of nodes have been found.

This function has a fast mode which runs if `fastElementParsing` = True and the number of nodes to define the element is less than or equal to 10. This assumes the element definition is not split across multiple lines. This should cover the most common Abaqus element types, as long as the pre-processor has not written the element data across more than 1 line.

**Returns**

A `Mesh` instance containing the data for the keyword block. Uses the element label as the key and the parsed dataline as the value.

**Return type**

`Mesh`

**`_parseDataGeneral()`**

Parses datalines without any special considerations.

This method parses any type of data. It splits the data lines and tries to turn each data item into their non-text type using `eval2()`. This function will split each data line on the “,” symbol and `eval2()` each item individually. All comments are stored in `comments`, or in the comments attribute if `_data`, if appropriate.

The other `_parseData*` functions will work similarly to `_parseDataGeneral()` (i.e. check if the dataline is a comment, split the dataline on commas to separate items, `eval2(item)`). The other functions will either take a shortcut in `eval2()` by specifying the expected data type or will perform some additional action(s) while looping through the data.

**Returns**

A list of lists containing the data for the keyword block.

**Return type**

`list`

**`_parseDataHeading()`**

Parses the datalines of \*HEADING keyword blocks.

Special method to parse the \*HEADING data lines. Simply returns the raw string as a list of lines without operating on the text.

**Returns**

A list of strings containing the data for the keyword block.

**Return type**

`list`

**`_parseDataNode()`**

Parses the datalines of \*NODE keyword blocks.

This is a special method for parsing \*NODE keyword blocks with data lines that must be exclusively integers or floats. All comments are stored in the appropriate `comments` section.

**Returns**

A `Mesh` instance that uses the node label as the key, and the parsed dataline as the value.

**Return type**

`Mesh`

**`_parseDataParameter()`**

Parses the datalines of \*PARAMETER keyword blocks.

This function parses the datalines of \*PARAMETER keyword blocks. In addition to filling `data` of the keyword block, this function will execute all the parameter functions and store the results in `pd` using the parameter name as the key. All comments are stored in `comments`.

**Returns**

A list of lists containing the data for the keyword block.

**Return type**

`list`

**`_parseDataRebarLayer()`**

Parses the datalines of \*REBAR LAYER keyword blocks.

This function parses the datalines of \*REBAR LAYER keyword blocks. In addition to filling `data` of the keyword block, this function will add the rebar layer names to `_namedRefs`. All comments are stored in `comments`.

**Returns**

A list of lists containing the data for the keyword block.

**Return type**

`list`

**`_parseDataSet()`**

Parses the \*NSET and \*ELSET keyword blocks.

This is a special method for parsing \*NSET and \*ELSET keyword blocks. For keyword blocks with 100 or more datalines, all items are assumed to be integers (i.e. labels). If items are not labels (i.e. set names), they will be treated as strings. This should increase the performance of the code for very large set keyword blocks, while hurting performance for sets defined by referencing existing sets (which should not have many datalines). For blocks with less than 100 datalines, `_parseDataGeneral()` will be called on lines, as this will avoid raising Exceptions in `eval2()` if a data item is not an integer. All comments are stored in `comments`.

**Returns**

A list of lists containing the data for the keyword block.



**Return type**

list

**\_parseDataSurface()**

parseDataSurface()

Parses \*SURFACE keyword blocks.

This is a special method for parsing \*SURFACE keyword blocks. It provides a fast method for TYPE=ELEMENT and TYPE=NODE, which should have much more data than the other types. It has an optimized path if `len(lines) > 100`. Otherwise, it simply calls `_parseDataGeneral()` on the data. All comments are stored in `comments`.

**Returns**

A list of lists containing the data for the keyword block.

**Return type**

list

**\_parseInputString(inputString="", parseKwLine=True)**

This function will split the `inputString` attribute on new lines, and will identify the keyword line (calling subfunctions to handle the tasks). It will populate the `name` and `parameter` attributes, and it will return a list of strings corresponding to the data lines.

If `_debug` is not True and `parseKwLine` is True, this function will set `inputString` to "" upon completion as a means to reduce memory consumption.

**Parameters**

- **inputString** (*str*) – A single string (with new line characters) composing the entirety of the keyword block. If `inputString` is not specified (thus ""), this function will parse `inputString`. Defaults to "".
- **parseKwLine** (*bool*) – If True, this function will parse the keyword line. Setting this to False will simply split the string into separate lines (used mainly for keyword blocks with no keyword line, which `inpKeyword` does internally to house data read from sub files). Defaults to True.

**Returns**

A list of strings corresponding to the datalines. The keyword line(s) will not be output if `parseKwLine = True`.

**Return type**

list

**\_parseKWLine(line)**

Populates `name` and `parameter` from the keyword line string. This function will also set `_parseData()` and `formatData()` based on `name`.

This function is meant to be called by `_parseInputString()`.

**Parameters**

**line** (*str*) – A string containing the entirety of the keyword line, but only a single line. `_parseInputString()` will handle parameters on subsequent lines for keyword lines which span multiple lines.

**\_parseSubData(subName, parentKwLine="")**

Populates the `data` attributed of a keyword block by reading from `subName`.

This function is meant to be called as part of `_parseData()`, as that function will handle some organizational operations. See that function for more details.

This function will only be called if `parseSubFiles` is True.

#### Parameters

- **subName** (*str*) – A string representing the sub-file from which to read. This should be taken from the INPUT parameter of the parent *inpKeyword* block.
- **parentKwLine** (*str*) – A string representing the keyword line of the parent *inpKeyword* block. This is only used to print some information, so it is not strictly necessary. Defaults to ‘’.

#### `__weakref__`

list of weak references to the object (if defined)

#### `_populateRebarLayerNamedRefs()`

`_populateRebarLayerNamedRefs()`

This function will populate `_namedRefs` with the \*REBAR LAYER names and a reference to this *inpKeyword*.

This function should only be called if you directly specify the `data` attribute of the block, and not if you created the block by parsing *inputString*. The parsing method will automatically perform the same operation.

#### `_setMiscInpKeywordAttrs()`

`_setMiscInpKeywordAttrs(parentInp)`

This function should be called after the `name` and `parameter` attributes have been set. It will set `_parseData()`, `formatData()`, and set and populate the appropriate items for `_inpItemsToUpdate`.

This function is called by `_parseKWLine()`, and thus automatically if *inpKeyword* is instantiated with the *inputString* parameter specified. It should be called manually if the *inpKeyword* is created without parsing *inputString*.

#### `_yieldInpRWItemsToUpdate()`

`_yieldInpRWItemsToUpdate():`

Yields the items in `_inpItemsToUpdate`.

The items to yield are set according to the keyword `name` in `_parseKWLine()`. They will be populated as part of the parsing function. These items are needed by the *inpRW.inpRW* for further operations, and will be added when the *inpKeyword* block is inserted into the *inpKeywordSequence*.

#### Yields

dict

#### `__getattr__(name)`

`__getattr__(name)`

Overrides the default behavior for cases where `data` and `comments` are not defined (i.e. when the keyword block reads data from a sub file).

See `__getattr__()` for more information.

#### Parameters

**name** (*str*) – The name of the attribute to retrieve.

#### Returns

A list containing the `data` or `comments` from `_data` and `_subdata`.

**Return type**

list

**Raises**

**AttributeError** – Will be raised if *inpKeyword* does not have attribute *name*, unless *name* is 'data' or 'comments'.

**\_\_str\_\_(includeSubData=True)**

Produces a string representation from the object in valid Abaqus formatting. All data of this *inpKeyword* instance will be fully expanded.

The *includeSubData* parameter defaults to True and cannot be accessed by the user when using built-in Python functions like `str` or `print()`. You must specifically call `block.__str__(includeSubData=False)` if you need to modify this parameter. `writeBlocks()` sets this to False to prevent writing the `_subdata` with the main keyword block; the `_subdata` will instead be written to the proper subfile.

**Parameters**

**includeSubData** (*bool*) – If True, the string of this *inpKeyword* will include all sub data. Defaults to True.

**Returns**

str

**\_\_repr\_\_(expand=False)**

Produces a string representation of the *inpKeyword* object.

By default, this will not expand *suboptions* or the entirety of *data* if `len(data)>10`. Set *expand* = True to display the entirety of *data* and all *suboptions* and their *suboptions*. Be aware that the block could contain an enormous amount of information.

**Parameters**

**expand** (*bool*) – If True, print the entire contents of the *inpKeyword*. Defaults to False.

**Returns**

str



## INPKEYWORDHELPER

### 22.1 Module Contents

This module contains functions meant to assist with creating *inpKeyword* blocks in parallel. This process is not currently providing much performance benefit (perhaps a 25% speedup when running with 4 cores vs. single core), so its usage is not recommended.

`inpKeywordHelper.inpKeywordInit(rawstringsin, Args)`

Initializes variables needed in each process when parsing keywords in parallel.

#### Parameters

- **rawstringsin** (*list*) – A list of strings. Each string should correspond to the text of an entire keyword block. The list will be turned into a global variable for the subprocesses.
- **Args** (*dict*) – Each item in Args will be converted to a global variable. This should be set to *inpKeywordArgs* in most cases.

`inpKeywordHelper.inpKeywordHelper(num)`

Creates an *inpKeyword* for `rawstrings[num]`, referencing the items passed to *inpKeywordInit()* via Args.

#### Parameters

- **num** (*int*) – An integer representing the item from `rawstrings` on which to operate.



## INPKEYWORDSEQUENCE

### 23.1 Module Contents

**class** `inpKeywordSequence.inpKeywordSequence`(*iterable=None, parentBlock=None*)

Bases: `list`

An *inpKeywordSequence* inherits from *list*, but it has some additional enhancements to make it more useful to *inpRW*. Mainly, an *inpKeywordSequence* automatically builds a reference to certain important information from each *inpKeyword* instance placed into the *inpKeywordSequence*, and it has a function which the main *inpRW* instance can call to retrieve this information from the main *inpKeywordSequence*, and all its children recursively. String formatting has also been modified to be more suited to this application.

**\_\_init\_\_**(*iterable=None, parentBlock=None*)

This class creates a blank *inpKeywordSequence* object, which inherits from the list type. It's main purpose is to hold *inpKeyword* objects, and it provides convenient methods for gathering important information from certain keyword blocks and making it available to the *inpRW* instance.

Creating a string representation of this class will create a useful summary of the keyword lines for each block, along with the *path* to the block, and the same information for the *suboptions* of each block if *printSubBlocks* = True.

For most users, the details of this class can be ignored. Simply create *inpKeyword* objects using one of the recommended options, and add the keyword blocks to the appropriate location.

In particular, this class has the following attributes which hold information needed by *inpRW*: *\_nd*, *\_ed*, *\_pd*, *\_namedRefs*, and *\_delayedPlaceBlocks*. The first four items are created when instantiating particular keyword blocks and stored in *\_inpItemsToUpdate*, and will be automatically inserted into the *inpKeywordSequence* if *inpRW* is on the correct insertion loop (specified by *\_keywordsDelayPlacingData*) when the keyword block is added to the *inpKeywordSequence*.

This class is also setup to pull these attributes from the same attributes from child *inpKeywordSequences* via *mergeSuboptionsGlobalData()*.

#### Parameters

- **iterable** (*list*) – A sequence of *inpKeyword* blocks with which to pre-populate the *inpKeywordSequence*. Defaults to []
- **parentBlock** (*inpKeyword*) – If the *inpKeywordSequence* instance is the *suboptions* attribute of a *inpKeyword* block, *parentBlock* should be set to this *inpKeyword* block. This is currently referenced only to report the path to access the `:class:~inpKeywordSequence`` instance. Defaults to None.

**parentBlock**

This will be None if the *inpKeywordSequence* is the *keywords* attribute, else it will refer to the *inpKeyword* block for which this item is the *suboptions* attribute.

**Type**

*inpKeyword*

**printSubBlocks**

If False, generating a string of this *inpKeywordSequence* will generate a string for each block in this sequence, along with the *path* to each block. If True, the *suboptions* for each block will also be fully expanded and represented.

**Type**

bool

**\_nd**

Stores the nodal data for all keyword blocks in this *inpKeywordSequence*, including that in the *suboptions* of blocks in this sequence. The user should interact with *nd*, which this attribute will eventually feed.

**Type**

*TotalMesh*

**\_ed**

Stores the element data for all keyword blocks in this *inpKeywordSequence*, including that in the *suboptions* of blocks in this sequence. The user should interact with *ed*, which this attribute will eventually feed.

**Type**

*TotalMesh*

**\_pd**

Stores the \*PARAMETER data for all keyword blocks in this *inpKeywordSequence*, including that in the *suboptions* of blocks in this sequence. The user should interact with *pd*, which this attribute will eventually feed.

**Type**

*csid*

**\_namedRefs**

Stores the named reference definitions for all keyword blocks in this *inpKeywordSequence*, including that in the *suboptions* of blocks in this sequence. The user should interact with *namedRefs*, which this attribute will eventually feed.

**Type**

*csid*

**\_delayedPlaceBlocks**

Contains the loop number on which specific information from keyword blocks should be placed, along with a list of all effected keyword blocks for that loop.

**Type**

*csid*

**\_grandchildBlocks**

contains the number of suboptions keyword blocks in each child block with suboptions.

**Type**

list



**append**(*x*, *place*=True, *loop*=None)

Appends *x* to the end of the *inpKeywordSequence*.

If *place* = True, will call *self.\_placeInpItemsToUpdate(x)* before running *list.append*. If *place* = False, *x* will be appended to *self.parentInp.\_delayedPlaceBlocks[loop]*, and it will also be inserted into the *inpKeywordSequence*.

#### Parameters

- **x** (*inpKeyword*) – The *inpKeyword* instance to append.
- **place** (*bool*) – If True, call *\_placeInpItemsToUpdate()* before appending *x*. Defaults to True.
- **loop** (*int*) – If *place* is False, *loop* must be specified. *x* will be appended to *self.parentInp.\_delayedPlaceBlocks[loop]*.

**extend**(*iterable*, *place*=True, *loop*=None)

Extends the *inpKeywordSequence* by appending all the items from *iterable*.

If *place* = True, will call *self.\_placeInpItemsToUpdate(x)* for *x* in *iterable* before running *list.extend*. If *place* = False, each item in *iterable* will be appended to *self.parentInp.\_delayedPlaceBlocks[loop]*, and they will also be inserted into the *inpKeywordSequence*.

#### Parameters

- **iterable** (*list*) – A sequence of *inpKeyword* instances with which to extend the *inpKeywordSequence*.
- **place** (*bool*) – If True, call *\_placeInpItemsToUpdate()* before inserting *iterable*. Defaults to True.
- **loop** (*int*) – If *place* is False, *loop* must be specified. Each item in *iterable* will be appended to *self.parentInp.\_delayedPlaceBlocks[loop]*.

**insert**(*i*, *x*, *place*=True, *loop*=None)

Inserts *x* into the *inpKeywordSequence* at position *i*.

If *place* = True, will call *self.\_placeInpItemsToUpdate(x)* before running *list.insert*. If *place* = False, *x* will be appended to *self.parentInp.\_delayedPlaceBlocks[loop]*, and it will also be inserted into the *inpKeywordSequence*.

#### Parameters

- **i** (*int*) – The index of the element before which to insert.
- **x** (*inpKeyword*) – The *inpKeyword* instance to insert.
- **place** (*bool*) – If True, call *\_placeInpItemsToUpdate()* before inserting *x*. Defaults to True.
- **loop** (*int*) – If *place* is False, *loop* must be specified. *x* will be appended to *self.parentInp.\_delayedPlaceBlocks[loop]*.

**mergeSuboptionsGlobalData()**

This function will add the contents of *\_nd*, *\_ed*, *\_pd*, *\_namedRefs*, and *\_delayedPlaceBlocks* from the *suboptions* of every block in the *inpKeywordSequence* recursively.

**\_placeInpItemsToUpdate**(*block*)

This function will place all items from *block.\_inpItemsToUpdate* into the appropriate attribute of the *inpKeywordSequence*.

This currently will place data into *:attr:\_namedRefs*, *:attr:\_ed*, *:attr:\_nd*, and *:attr:\_pd*.

**Parameters**

**block** (inpKeyword) –

## 24.1 Module Contents

The *inpRWErrors* module contains the custom exception types used throughout *inpRW*.

**exception** *inpRWErrors.inpRWErrors*(*args=None*)

Bases: *Exception*

The base class for all custom Exception types used by *inpRW*.

**\_\_init\_\_**(*args=None*)

Initializes the exception. *args* should be a *tuple*.

*inpRWErrors* is meant to be subclassed, not used directly. It provides a common base class for the other exceptions in this module.

Example usage:

```
>>> from inpRWErrors import inpRWErrors
>>> raise inpRWErrors
Traceback (most recent call last):
...
inpRWErrors.inpRWErrors: None
```

### Parameters

**args** (*tuple*) – A *tuple* of arguments which will be passed to the class.

**\_\_str\_\_**()

Creates a string error message of the object using the default Exception *\_\_str\_\_*() .

**\_\_weakref\_\_**

list of weak references to the object (if defined)

**exception** *inpRWErrors.KeywordNotFoundError*(*args=None*)

Bases: *inpRWErrors*

This class is a blank Exception and is raised by *findKeyword()* to break out of loops if the particular keyword cannot be found in *keywords*.

---

### Example

```
>>> from inpRWErrors import KeywordNotFoundError
>>> raise KeywordNotFoundError
Traceback (most recent call last):
...
inpRWErrors.KeywordNotFoundError: None
```

---

**exception** `inpRWErrors.DecimalInfError(args=None)`

Bases: `inpRWErrors`

This class is a blank Exception and is raised by `__new__()` if the input string is 'INF'.

---

#### Example

```
>>> from inpRWErrors import DecimalInfError
>>> raise KeywordNotFoundError
Traceback (most recent call last):
...
inpRWErrors.KeywordNotFoundError: None
```

---

**\_\_str\_\_()**

Creates a string error message of the object using the default Exception `__str__()`.

**exception** `inpRWErrors.ElementIncorrectNodeNumError(elementType, nodeNum)`

Bases: `inpRWErrors`

Raised when an element has been assigned the incorrect number of nodes for its element type.

**\_\_init\_\_**(*elementType*, *nodeNum*)

Sets the arguments which will be used in the Error message.

---

#### Example

```
>>> from inpRWErrors import ElementIncorrectNodeNumError
>>> raise ElementIncorrectNodeNumError('C3D8', 8)
Traceback (most recent call last):
...
inpRWErrors.ElementIncorrectNodeNumError: ERROR! An element of type C3D8 must_
↳have 8 nodes.
```

---

If this exception is raised without the proper arguments, a `TypeError` will instead be raised:

```
>>> raise ElementIncorrectNodeNumError
Traceback (most recent call last):
...
TypeError: ...__init__() missing 2 required positional arguments: 'elementType' _
↳and 'nodeNum'
```

---

#### Parameters

- **elementType** (*str*) –
- **nodeNum** (*int*) –

`__str__()`

Produces the string error message.

**exception** `inpRWEErrors.ElementTypeMismatchError`(*otherEltype*, *meshElementType*)

Bases: `inpRWEErrors`

Raised when an `Element` is inserted into a `MeshElement` class, and `eltype` parameter does not match `eltype` parameter.

`__init__`(*otherEltype*, *meshElementType*)

Sets the arguments which will be used in the Error message.

Example:

```
>>> from inpRWEErrors import ElementTypeMismatchError
>>> raise ElementTypeMismatchError('C3D8R', 'C3D8')
Traceback (most recent call last):
...
inpRWEErrors.ElementTypeMismatchError: ERROR! An element of type C3D8R
↳ cannot be added to a MeshElement with eltype C3D8
```

If this exception is raised without the proper arguments, a `TypeError` will instead be raised:

```
>>> raise ElementTypeMismatchError
Traceback (most recent call last):
...
TypeError: ...__init__() missing 2 required positional arguments: 'otherEltype'
↳ and 'meshElementType'
```

#### Parameters

- **otherEltype** (*str*) –
- **meshElementType** (*str*) –

`__str__()`

Produces the string error message.



## 25.1 Module Contents

**class** `inpString.inpString(inputStr, ss=False)`

Bases: `str`

An `inpString` object behaves like a regular `str`, except it will internally store the string value without any leading or trailing spaces (Abaqus is space-insensitive regarding leading or trailing spaces in names). This class will track the leading and trailing spaces using interned string patterns, which will be used to reproduce the original string exactly.

**static** `__new__(inputStr, ss=False)`

`__init__(inputStr, ss)`

Creates the `inpString` instance.

This is meant to operate on text which corresponds to a single string item. For example, a set name, a parameter name, etc. Thus, in most cases `inputStr` should be a single word with surrounding whitespace. The only time `inputStr` should contain a space between alphanumeric characters is when the value of interest is a quoted string and includes spaces between words.

Example usage:

```
>>> from inpString import inpString
>>> inpString(' Nodeset-1')
inpString(' Nodeset-1', False)
```

If we desire to remove trailing and leading spaces, we can set `ss = True`:

```
>>> inpString(' Nodeset-1', ss=True)
inpString('Nodeset-1', True)
```

### Parameters

- **inputStr** (`str`) – The unformatted string for which the `inpString` will be created.
- **ss** (`bool`) – If True, leading and trailing spaces will be removed from `inputStr`, and not reinserted when producing the output string.

### ss

If True, leading and trailing spaces will be permanently removed from `inputStr`. Defaults to False.

### Type

`bool`

**\_value**

Stores the non-whitespace information of *inputStr*. Defaults to None

**\_formatStr**

Stores the whitespace information of *inputStr*, with placeholder symbols (“%s”) where *\_value* should be reinserted. Defaults to None

**\_evalString(inputStr)**

*\_evalString(inputStr)*

Evaluates the input string and performs some formatting. This will separate whitespace from alphanumeric characters, but it will leave spaces between alphanumeric characters.

This will set *\_value* and *\_formatStr*, but the function has no direct return.

Here’s an example of the *\_value* and *\_formatStr* attributes for a simple case:

```
>>> from inpString import inpString
>>> s1 = inpString(' Nodeset-1')
>>> s1._value
'Nodeset-1'
>>> s1._formatStr
' %s'
```

Here are the same attributes when *ss* = True:

```
>>> s2 = inpString(' Nodeset-1', ss=True)
>>> s2._value
'Nodeset-1'
>>> s2._formatStr
'%s'
```

Finally, if we have a quoted string with spaces, the spaces inside the quotes are included with *\_value* because they’re meaningful. The spaces outside the quotes are tracked in *\_formatStr* as usual:

```
>>> s3 = inpString(' "Nodeset Name with Spaces"')
>>> s3._value
'"Nodeset Name with Spaces"'
>>> s3._formatStr
' %s'
```

**\_outstr()**

Generates the output string. *\_value* is subbed into *\_formatStr*. Should not need to be called by the user directly.

Example usage:

```
>>> from inpString import inpString
>>> s = inpString(' Nodeset-1')
>>> s._outstr()
' Nodeset-1'
```

This function can also handle cases where *inputStr* to *\_\_init\_\_()* was only whitespace:

```
>>> s = inpString(' ')
>>> s._outstr()
'
```



**Returns**

The string in the original formatting.

**Return type**

str

**\_\_str\_\_()**

Calls `_outstr()` to create a string of object in the original formatting.

Example usage:

```
>>> from inpString import inpString
>>> print(inpString(' Nodeset-1'))
Nodeset-1
```

**Return type**

str

**\_\_repr\_\_()**

Creates a string representation which can be used to recreate the object.

Example usage:

```
>>> from inpString import inpString
>>> inpString(' Nodeset-1')
inpString(' Nodeset-1', False)
```

**Return type**

str

**\_\_getnewargs\_\_()**

`__getnewargs__()`:

This function is required so the class can pickle/unpickle properly. This enables multiprocessing, and should not need to be called by the user.

Example usage:

```
>>> from inpString import inpString
>>> s = inpString(' Nodeset-1')
>>> s.__getnewargs__()
(' Nodeset-1', False)
```

**\_\_weakref\_\_**

list of weak references to the object (if defined)



## 26.1 Module Contents

The `mesh` module contains classes that store node and element data.

```
class mesh.Node(dataline="", label=None, data=None, elements=None, _joinS=', ', _ParamInInp=False,
                preserveSpacing=True, useDecimal=True)
```

Bases: `object`

The Node class stores nodal information from a single node (i.e. one dataline from a \*NODE keyword block).

```
__init__(dataline="", elements=None, _ParamInInp=False, preserveSpacing=True, useDecimal=True)
```

```
__init__(label=None, data=None, elements=None, _joinS=', ') → None
```

Initializes attributes of the Node.

In most cases, this will be used by `_parseDataNode()`. That function will handle creating the inputs to this class from the string corresponding to the node dataline. See that function for an example of instantiating this class. If you are instantiating this class manually, there are two recommended options.

First, specify the `dataline` parameter and the other related parameters if you know them prior to creating the `Node` instance and/or do not wish to use the default values. `dataline` will be parsed, and `label` and `data` will be populated from the parsed information.

Here's an example using the default settings, which will use `inpInt`, `inpDecimal.inpDecimal`, and `inpString` to store data items and preserve the exact spacing:

```
>>> from mesh import Node
>>> dataline = '    21, 70.00,    0.00'
>>> node = Node(dataline)
>>> node
Node(label=inpInt('    21'), data=[inpInt('    21'), inpDecimal(' 70.00'),
↳ inpDecimal('    0.00')], elements=[])
```

The `repr()` of an `Node` instance is consistent across the different methods to instantiate the class. If you wish to see how the `Node` will be written out, you can use a function to call its string representation. For example:

```
>>> print(node)
    21, 70.00,    0.00
```

Here's an example of turning off the mechanisms for preserving the exact spacing (for a performance benefit):

```
>>> node = Node(dataline, preserveSpacing=False, useDecimal=False)
>>> print(node)
21, 70.0, 0.0
```

We can also specify a custom separator to join the parsed strings when we create a string of the node. This will only be applied if *preserveSpacing* and *useDecimal* are False. Example:

```
>>> node = Node(dataline, _joinS = ',__', preserveSpacing=False,
↳useDecimal=False)
>>> print(node)
21,__70.0,__0.0
```

If an item in *dataline* could be a \*PARAMETER reference, either *\_ParamInInp* or *preserveSpacing* must be True, or else a *ValueError* will be raised. Example:

```
>>> dataline = ' 21, <coord>, 0.00'
>>> node = Node(dataline, preserveSpacing=False, useDecimal=False)
Traceback (most recent call last):
...
ValueError: could not convert string to float: ' <coord>'
```

The second option is to specify *label* and *data* instead of *dataline*. Those items should be formatted properly prior to instantiating *Node*. Example:

```
>>> label = 21
>>> data = [21, 70.0, 0.00]
>>> node = Node(label=label, data=data)
>>> print(node)
21, 70.0, 0.0
```

Using either method, *elements* can be specified if the user knows the list of elements which connect to the node. In most cases, this should not be specified, and the elements connected to the node should instead be added using *setConnectedNodes()*.

### Parameters

- **dataline** (*str*) – The string which contains the entire dataline for the node. Will be parsed to populate the *node's* attributes. Defaults to ''
- **label** (*int*) – The label for the node. Defaults to None.
- **data** (*list*) – The data for the node. This will be a parsed dataline. The format should be [*int*, *float*, *float*, ...]. Defaults to None.
- **elements** (*list*) – The elements which are connected to the *Node* instance. Defaults to None.
- **\_joinS** (*str*) – The string used to join the data items together when creating strings of the *Node* instance. Defaults to ','.
- **\_ParamInInp** (*bool*) – Indicates if \*PARAMETER is in the input file, which would mean that the dataline items could be strings instead of exclusively integers. Defaults to False.
- **preserveSpacing** (*bool*) – If True, the exact spacing of all items will be preserved by using *inpInt*. In most cases when creating new *Element* instances, you will want to use the same value that the rest of the input file was parsed with. Defaults to True.

- **useDecimal** (*bool*) – If True, any floating point numbers will be parsed as `Decimal` or `inpDecimal`. In most cases when creating new `Element` instances, you will want to use the same value that the rest of the input file was parsed with. Defaults to True.

#### **\_ParamInInp**

Indicates if `*PARAMETER` is in the input file. If True, data items could be references to `*PARAMETER` functions, and the datalines might be strings, so the `:func:'~eval2.eval2'` function must be used when parsing data items instead of assuming they must be integers. Defaults to False.

**Type**  
`bool`

#### **preserveSpacing**

If True, the exact spacing of all items will be preserved by using `inpInt` instead of `int`. `_joinS` will be set to `' '` if `preserveSpacing` is True, as the spacing will be stored in the data objects. Defaults to True.

**Type**  
`bool`

#### **useDecimal**

If True, any floating point numbers will be parsed as `Decimal` or `inpDecimal`. Defaults to True.

#### **\_joinS**

The string used to join data items together when creating a string for the entire dataline. Defaults to `' '`, which will be sufficient for most cases.

**Type**  
`str`

#### **label**

The node label.

**Type**  
`int`

#### **data**

A list containing the parsed data items.

**Type**  
`list`

#### **elements**

Will store the elements that reference this `Node`. This is populated by `Element.setConnectedNodes()`.

**Type**  
`list`

#### **\_parseDataLine(dataline)**

Populates the `Node` instance by parsing the `dataline` string.

This function will be called automatically when `Node` is initialized if `datalines` is specified.

Example usage:

```
>>> from mesh import Node
>>> node = Node()
>>> node._parseDataLine(' 21, 70.00, 0.00')
>>> node
Node(label=inpInt(' 21'), data=[inpInt(' 21'), inpDecimal(' 70.00'),
↳ inpDecimal(' 0.00')], elements=[])
```

**Parameters**

**dataline** (*str*) – A string containing the information for the entire dataline.

**\_mergeElementData**(*other*)

Currently unused

**updateElements**(*ed*)

Checks if each element in *elements* exists in *ed*. Returns 0 if *elements* is empty, 1 if elements have been removed, or 2 if no changes were made.

**Parameters**

**ed** (*TotalMesh*) – This is the *TotalMesh* instance containing elements in which to check for *elements*.

**Returns**

0 if *elements* is now empty, 1 if *elements* was modified, 2 if *elements* is unchanged.

**Return type**

*int*

**\_\_repr\_\_**()

Produces a repr of the Node.

---

**Example**

```
>>> from mesh import Node
>>> Node(' 21, 70.00, 0.00')
Node(label=inpInt(' 21'), data=[inpInt(' 70.00'),
↳ inpDecimal(' 0.00')], elements=[])
```

**\_\_str\_\_**()

Produces a str of the dataline for the Node.

---

**Example**

```
>>> from mesh import Node
>>> node = Node(' 21, 70.00, 0.00')
>>> print(node)
21, 70.00, 0.00
```

**\_\_weakref\_\_**

list of weak references to the object (if defined)

**class** *mesh.Element*(*eltype*, *numNodes=None*, *datalines=None*, *label=None*, *data=None*, *\_lineEnd=None*, *\_npl=None*, *\_joinS=' '*, *\_ParamInInp=False*, *preserveSpacing=True*, *useDecimal=True*, *\_checkNumNodes=True*, *\_nl='\n'*)

Bases: *object*

The Element class stores element information from a single element (i.e. one dataline from a \*ELEMENT keyword block).

**\_\_init\_\_**(*eltype*, *numNodes=None*, *datalines=None*, *'*, *\_ParamInInp=False*, *preserveSpacing=True*, *useDecimal=True*, *\_checkNumNodes=True*, *\_nl='\n'*)

`__init__(eltype, numNodes=None, label=None, data=None, _lineEnd=None, _npI=None, _joinS=', ') → None`

Initializes attributes of the Element.

In most cases, this will be used by `_parseDataElement()`. That function will handle creating the inputs to this class from the string corresponding to the element dataline. See that function for more details. If you are instantiating this class manually, you can omit specifying the private parameters. The default values should be sufficient.

All parameters are optional except `eltype`, but there are two recommend methods to create an instance of this class. The first method is to specify `datalines` and allow `Element` to parse these datalines to populate `label` and `data`. Here is an example of this use case with a CPS4R element:

```
>>> from mesh import Element
>>> datalines = ' 1, 1, 2, 102, 101'
>>> element = Element(eltype='CPS4R', datalines=datalines)
>>> element
Element(label=inpInt(' 1'), data=[inpInt(' 1'), inpInt(' 1'), inpInt(' 2'),
↳inpInt(' 102'), inpInt(' 101')], eltype='CPS4R', numNodes=4)
```

The `repr()` of an `Element` instance is consistent across the different methods to instantiate the class. If you wish to see how the `Element` will be written out, you can use a function to call its string representation. For example:

```
>>> print(element)
1, 1, 2, 102, 101
```

Here's a similar example with an element defined over multiple lines:

```
>>> datalines = [' 1, 16, 4, 24, 57, 13, 1, 32, 66, 92, 91, 90,
↳89, 93, 94, 95,',
... ' 96, 98, 97, 99, 100']
>>> Element(eltype='C3D20', datalines=datalines)
Element(label=inpInt(' 1'), data=[inpInt(' 1'), inpInt(' 16'), inpInt(' 4'),
↳inpInt(' 24'), inpInt(' 57'), inpInt(' 13'), inpInt(' 1'), inpInt(' 32
↳'), inpInt(' 66'), inpInt(' 92'), inpInt(' 91'), inpInt(' 90'), inpInt('
↳89'), inpInt(' 93'), inpInt(' 94'), inpInt(' 95'), inpInt(' 96'),
↳inpInt(' 98'), inpInt(' 97'), inpInt(' 99'), inpInt(' 100')], eltype='C3D20
↳', numNodes=20)
```

We can also specify `datalines` as a single string with newline characters. Example:

```
>>> datalines = ' 1, 16, 4, 24, 57, 13, 1, 32, 66, 92, 91, 90, 89,
↳ 93, 94, 95,\n 96, 98, 97, 99, 100'
>>> Element(eltype='C3D20', datalines=datalines)
Element(label=inpInt(' 1'), data=[inpInt(' 1'), inpInt(' 16'), inpInt(' 4'),
↳inpInt(' 24'), inpInt(' 57'), inpInt(' 13'), inpInt(' 1'), inpInt(' 32
↳'), inpInt(' 66'), inpInt(' 92'), inpInt(' 91'), inpInt(' 90'), inpInt('
↳89'), inpInt(' 93'), inpInt(' 94'), inpInt(' 95'), inpInt(' 96'),
↳inpInt(' 98'), inpInt(' 97'), inpInt(' 99'), inpInt(' 100')], eltype='C3D20
↳', numNodes=20)
```

If we set `preserveSpacing`, `useDecimal`, and `_ParamInInp` all to `False`, we gain some performance at the expense of some formatting accuracy. See the difference below:

```

>>> datalines = ' 1, 16, 4, 24, 57, 13, 1, 32, 66, 92, 91, 90, 89,
↳ 93, 94, 95,\n 96, 98, 97, 99, 100'
>>> print(Element(eltype='C3D20', datalines=datalines, preserveSpacing=False,
↳ useDecimal=False))
1, 16, 4, 24, 57, 13, 1, 32, 66, 92, 91, 90, 89, 93, 94, 95,
96, 98, 97, 99, 100
>>> print(Element(eltype='C3D20', datalines=datalines))
1, 16, 4, 24, 57, 13, 1, 32, 66, 92, 91, 90, 89, 93, 94, 95,
96, 98, 97, 99, 100

```

The second method is to instead specify *label* and *data* directly. We will need to get the data into a suitable form prior to inserting it into the *Element* instance. You will use this option most often if you are creating new elements and do not start with a string containing the entire element definition. Example usage:

```

>>> label = 1
>>> data = [1, 16, 4, 24, 57, 13, 1, 32, 66, 92, 91, 90, 89, 93, 94, 95, 96, 98,
↳ 97, 99, 100]
>>> element = Element(eltype='C3D20', label=label, data=data)
>>> print(element)
1, 16, 4, 24, 57, 13, 1, 32, 66, 92, 91, 90, 89, 93, 94, 95,
96, 98, 97, 99, 100

```

If you don't specify the number of nodes to include per each line via *\_np11*, there will be 16 items per line.

We can also specify the strings used to join data items (*\_joinS*), and a string to include at the end of each line (*\_lineEnd*). This should normally be whitespace characters, but different characters are used here for illustrative purposes:

```

>>> element = Element(eltype='C3D20', label=label, data=data, _lineEnd=['++',
↳ None], _joinS = ',-')
>>> print(element)
1,-16,-4,-24,-57,-13,-1,-32,-66,-92,-91,-90,-89,-93,-94,-95,++
96,-98,-97,-99,-100

```

### Parameters

- **eltype** (*str*) – The element type name. Should be a valid Abaqus element type name, or the name of a user-element.
- **numNodes** (*int*) – The number of nodes the element needs. If not specified, this will be retrieved from *\_elementTypeDictionary[eltype]*. This should only be specified if an element type is not in that dictionary, which will mainly be for user or substructure elements.
- **datalines** (*list*, *str*) – A list of strings representing the datalines for the *Element*. This can also be a string representing the entirety of the data for the element, with new line characters if the definition is spread across multiple lines. Defaults to None.
- **label** (*int*) – The label for the element. Defaults to None.
- **data** (*list*) – The data for the element. This will be a parsed dataline. The format should be [*int*, *int*, ...].
- **\_lineEnd** (*str*, *list*) – Holds any additional whitespace characters to include at the end of the dataline. Mainly used to exactly reproduce the string representation of the keyword block. Defaults to None. If the data must be printed across multiple lines, *\_lineEnd* will be expanded to a list if necessary, with each item in the list corresponding to the old value



of `_lineEnd`. If a `_lineEnd` item is `None`, nothing will be added to the end of that line. If a `_lineEnd` item is a string, the `_lineEnd` item will follow a comma on the effected line.

- **`_npl1`** (*list*) – Nodes Per Line List, controls how many nodes should be printed on each line when converting the `Element` to a string. The default value of `None` will cause the maximum number of nodes to be printed on each line (up to 15 for line 0, and up to 16 for all subsequent lines).
- **`_joinS`** (*str*) – The string used to join the data items together when creating strings of the `Element` instance. Defaults to `None`, which will use `‘, ‘`. If `preserveSpacing` is `True` and `datalines` is not `None`, `‘, ‘` will be used.
- **`_ParamInInp`** (*bool*) – Indicates if `*PARAMETER` is in the input file, which would mean that the `datalines` items could be strings instead of exclusively integers. Defaults to `False`.
- **`preserveSpacing`** (*bool*) – If `True`, the exact spacing of all items will be preserved by using `inpInt`. In most cases when creating new `Element` instances, you will want to use the same value that the rest of the input file was parsed with. Defaults to `True`.
- **`useDecimal`** (*bool*) – If `True`, any floating point numbers will be parsed as `Decimal` or `inpDecimal`. In most cases when creating new `Element` instances, you will want to use the same value that the rest of the input file was parsed with. Defaults to `True`.
- **`_checkNumNodes`** (*bool*) – If `True`, the number of nodes parsed from `datalines` will be checked against `numNodes`. If the `Element` has the incorrect number of nodes, an `ElementIncorrectNodeNumError` will be raised. Defaults to `True`.
- **`_nl`** (*str*) – Used to split `datalines` if it is a string. Defaults to `‘n’`.

### **eltype**

Indicates the element type name. Should be a valid Abaqus element type.

#### **Type**

`str`

### **\_checknumNodes**

If `True`, the number of nodes parsed from `datalines` will be checked against the number in `numNodes`. Defaults to `True`.

#### **Type**

`bool`

### **numNodes**

The number of nodes needed for the `Element`. Can be specified by the user, or will be retrieved from `_elementTypeDictionary[eltype]`.

#### **Type**

`int`

### **\_ParamInInp**

Indicates if `*PARAMETER` is in the input file. If `True`, data items could be references to `*PARAMETER` functions, and the `datalines` might be strings, so the `:func:~eval2.eval2` function must be used when parsing data items instead of assuming they must be integers. Defaults to `False`.

#### **Type**

`bool`

### **preserveSpacing**

If `True`, the exact spacing of all items will be preserved by using `inpInt` instead of `int`. Defaults to `True`.

**Type**  
bool

**\_nl**

Used to split *datalines* if it is a string. Defaults to 'n', which should be correct for most cases.

**Type**  
str

**useDecimal**

If True, any floating point numbers will be parsed as *Decimal* or *inpDecimal*. Defaults to True.

**Type**  
bool

**\_joinS**

The string used to join the data items together when creating strings of the *Element* instance.

**Type**  
str

**label**

The element label.

**Type**  
int

**data**

The data for the element. This will be a parsed dataline. The format should be [int, ...].

**Type**  
list

**\_npl1**

Nodes Per Line List, controls how many nodes will be printed on each line when converting *Element* to a string.

**Type**  
list

**\_lineEnd**

Holds any additional whitespace characters to include at the end of the dataline.

**Type**  
str

**checkNumberNodes()**

checkNumberNodes()

This function will check if the number of nodes in *data* matches that specified in *numNodes*.

It raises an *ElementIncorrectNodeNumError* if the number of nodes does not match the element type requirements. Otherwise, it does nothing. This function will be called by *\_parseDataLines()* if *\_checknumNodes* is True (thus, it will be called if *Element* is initialized with *datalines* and the default settings). It should be called manually if the *Element* is populated in another manner.

---

**Example**

```

>>> from mesh import Element
>>> element = Element(eltype='CPS4R', label=1, data=[1, 1, 2, 102, 101, 103])
>>> element.checkNumberNodes()
Traceback (most recent call last):
...
inpRWEErrors.ElementIncorrectNodeNumError: ERROR! An element of type CPS4R must
↳ have 4 nodes.

```

### Raises

*ElementIncorrectNodeNumError* –

### `_parseDataLines(datalines)`

Populates the *Element* instance by parsing the strings in *datalines*.

This function will also call *checkNumberNodes()* if *\_checknumNodes* = True. It is called automatically if *Element* is instantiated with the *datalines* argument.

Example usage:

```

>>> from mesh import Element
>>> element = Element(eltype='CPS4R')
>>> element._parseDataLines('1, 1, 2, 102, 101')
>>> print(element)
1, 1, 2, 102, 101

```

### Parameters

**datalines** (*list*) – A list of strings containing the information for the entire dataline.

### `_parseDataLine(dataline, con=False)`

Parses an individual \*ELEMENT dataline. Meant to be used by *\_parseDataLines()*.

Example usage:

```

>>> from mesh import Element
>>> dataline1 = ' 1, 16, 4, 24, 57, 13, 1, 32, 66, 92, 91, 90, 89,
↳ 93, 94, 95,'
>>> dataline2 = ' 96, 98, 97, 99, 100'
>>> element = Element(eltype='C3D20')
>>> element._parseDataLine(dataline1)
>>> element
Element(label=inpInt(' 1'), data=[inpInt(' 1'), inpInt(' 16'), inpInt(' 4'),
↳ inpInt(' 24'), inpInt(' 57'), inpInt(' 13'), inpInt(' 1'), inpInt(' 32
↳ '), inpInt(' 66'), inpInt(' 92'), inpInt(' 91'), inpInt(' 90'), inpInt('
↳ 89'), inpInt(' 93'), inpInt(' 94'), inpInt(' 95')], eltype='C3D20',
↳ numNodes=20)
>>> element._parseDataLine(dataline2, con=True)
>>> element
Element(label=inpInt(' 1'), data=[inpInt(' 1'), inpInt(' 16'), inpInt(' 4'),
↳ inpInt(' 24'), inpInt(' 57'), inpInt(' 13'), inpInt(' 1'), inpInt(' 32
↳ '), inpInt(' 66'), inpInt(' 92'), inpInt(' 91'), inpInt(' 90'), inpInt('
↳ 89'), inpInt(' 93'), inpInt(' 94'), inpInt(' 95'), inpInt(' 96'),

```

(continues on next page)

(continued from previous page)

```

↳ inpInt(' 98'), inpInt(' 97'), inpInt(' 99'), inpInt(' 100')], eltype='C3D20
↳ ', numNodes=20)

```

**Parameters**

- **dataline** (*str*) – The string containing the information for one line of data.
- **con** (*bool*) – If True, this function will treat *dataline* as a continuation of the previous dataline (i.e. all items represent node labels, instead of an element label and then node labels). Defaults to False.

**\_processDataLinesInput(*datalines*)**

If *datalines* is a string, split *datalines* on *\_nl*, which will convert *datalines* to a string.

This is called automatically by *\_parseDataLines()*, so the user will likely never need to call it directly.

Example usage:

```

>>> from mesh import Element
>>> element = Element(eltype='CPS4R')
>>> element._processDataLinesInput(' 1, 1, 2, 102, 101')
[' 1, 1, 2, 102, 101']
>>> element._processDataLinesInput(' 1, 16, 4, 24, 57, 13, 1, 32, 66,
↳ 92, 91, 90, 89, 93, 94, 95,\n 96, 98, 97, 99, 100')
[' 1, 16, 4, 24, 57, 13, 1, 32, 66, 92, 91, 90, 89, 93, 94, 95,
↳ ', ' 96, 98, 97, 99, 100']

```

**Parameters**

**datalines** (*str*, *list*) – A string or list of strings. If a string, will be converted to a list by splitting on *\_nl*.

**Returns**

list

**setConnectedNodes(*nd*)**

This function will find each node in *nd* corresponding to the node labels in *data* and append *label* to the *elements*.

**Parameters**

**nd** (*TotalMesh*) – The TotalMesh instance containing the nodes for the input file.

**\_\_repr\_\_()**

Produces a repr of the dataline for the Element.

Example usage:

```

>>> from mesh import Element
>>> Element(eltype='CPS4R', datalines=' 1, 1, 2, 102, 101')
Element(label=inpInt(' 1'), data=[inpInt(' 1'), inpInt(' 1'), inpInt(' 2'),
↳ inpInt(' 102'), inpInt(' 101')], eltype='CPS4R', numNodes=4)

```

**Returns**

string

**\_\_str\_\_()**

Produces a str of the dataline for the Element.

Example usage:

```
>>> from mesh import Element
>>> print(Element(eltype='CPS4R', datalines='1, 1, 2, 102, 101'))
1, 1, 2, 102, 101
```

**Returns**

string

**\_\_weakref\_\_**

list of weak references to the object (if defined)

**class** mesh.Mesh(*data=None*)

Bases: *csid*

This class is a *csid* used to store the data from a node *keyword block*.

**\_\_init\_\_**(*data=None*)

Initializes the class. In most cases, users should not directly instantiate this class. Rather, they should create a \*NODE *inpKeyword* instance and allow that keyword block to handle the logistics.

Example usage:

```
>>> from mesh import Node, Mesh
>>> node1 = Node(' 1 , 0.00, 0.00')
>>> node2 = Node(' 21, 70.00, 0.00')
>>> Mesh([[i.label, i] for i in [node1, node2]])
csid(inpInt(' 1 '): Node(label=inpInt(' 1 '), data=[inpInt(' 1 '),
↳ inpDecimal(' 0.00'), inpDecimal(' 0.00')], elements=[]), inpInt(' 21'):
↳ Node(label=inpInt(' 21'), data=[inpInt(' 21'), inpDecimal(' 70.00'),
↳ inpDecimal(' 0.00')], elements=[]))
```

**Parameters**

**data** – *data* should be a format that can generate a *dictionary*. Defaults to None, which will produce a blank object.

**\_\_repr\_\_()**

Uses the default `__repr__` method from *dict*.

Example usage:

```
>>> from mesh import Node, Mesh
>>> node1 = Node(' 1 , 0.00, 0.00')
>>> node2 = Node(' 21, 70.00, 0.00')
>>> Mesh([[i.label, i] for i in [node1, node2]])
csid(inpInt(' 1 '): Node(label=inpInt(' 1 '), data=[inpInt(' 1 '),
↳ inpDecimal(' 0.00'), inpDecimal(' 0.00')], elements=[]), inpInt(' 21'):
↳ Node(label=inpInt(' 21'), data=[inpInt(' 21'), inpDecimal(' 70.00'),
↳ inpDecimal(' 0.00')], elements=[]))
```

`__str__()`

Uses the default `__str__` method from `dict`.

Example usage:

```
>>> from mesh import Node, Mesh
>>> node1 = Node(' 1 , 0.00, 0.00')
>>> node2 = Node(' 21, 70.00, 0.00')
>>> print(Mesh([[i.label, i] for i in [node1, node2]]))
{
  1 :      1 , 0.00, 0.00
  21:     21, 70.00, 0.00
}
```

**class** `mesh.MeshElement`(*eltype*, *data=None*)

Bases: `Mesh`

`MeshElement` is identical to `Mesh`, except that it checks to make sure that every item added to it has the same element type.

`__init__`(*eltype*, *data=None*)

`__init__`(*eltype*, *data=None*)

Initializes the class. In most cases, users should not directly instantiate this class. Rather, they should create a \*ELEMENT *inpKeyword* instance and allow that keyword block to handle the logistics.

Example usage:

```
>>> from mesh import Element, MeshElement
>>> element1 = Element(eltype='C3D20', datalines=' 1, 16, 4, 24, 57, 13, \n
↳ 1, 32, 66, 92, 91, 90, 89, 93, 94, 95,\n      96, 98, 97, 99, 100
↳ ')
>>> element2 = Element(eltype='C3D20', datalines=' 2, 15, 16, 57, 58, 14, \n
↳ 13, 66, 65, 103, 89, 102, 101, 104, 96, 105,\n      106, 107, 98, 100, 108
↳ ')
>>> mesh = MeshElement(eltype='C3D20', data=[[i.label, i] for i in [element1,
↳ element2]])
>>> mesh
csid(inpInt(' 1'): Element(label=inpInt(' 1'), data=[inpInt(' 16
↳ '), inpInt(' 4'), inpInt(' 24'), inpInt(' 57'), inpInt(' 13'), inpInt('
↳ 1'), inpInt(' 32'), inpInt(' 66'), inpInt(' 92'), inpInt(' 91'), inpInt(
↳ ' 90'), inpInt(' 89'), inpInt(' 93'), inpInt(' 94'), inpInt(' 95'),
↳ inpInt(' 96'), inpInt(' 98'), inpInt(' 97'), inpInt(' 99'), inpInt('
↳ 100')], eltype='C3D20', numNodes=20), inpInt(' 2'): Element(label=inpInt(' 2
↳ '), data=[inpInt(' 2'), inpInt(' 15'), inpInt(' 16'), inpInt(' 57'),
↳ inpInt(' 58'), inpInt(' 14'), inpInt(' 13'), inpInt(' 66'), inpInt(' 65
↳ '), inpInt(' 103'), inpInt(' 89'), inpInt(' 102'), inpInt(' 101'), inpInt('
↳ 104'), inpInt(' 96'), inpInt(' 105'), inpInt(' 106'), inpInt(' 107'),
↳ inpInt(' 98'), inpInt(' 100'), inpInt(' 108')], eltype='C3D20', numNodes=20))
```

### Parameters

- **eltype** (*str*) – The string representing the Abaqus element type name.
- **data** – *data* should be a format that can generate a `dictionary`. Defaults to `None`, which will produce a blank object.

**eltype**

A string representing the element type name

**Type**

str

**\_\_setitem\_\_(label, element)**

Checks if *element.eltype* matches *eltype*. If it does, set `MeshElement[label] = element`, else raise *ElementTypeMismatchError*.

Correct usage, the *eltype* of the *Element* matches that of the *MeshElement*:

```
>>> from mesh import Element, MeshElement
>>> element1 = Element(eltype='C3D20', datalines=' 1, 16, 4, 24, 57, 13, \n
↳ 1, 32, 66, 92, 91, 90, 89, 93, 94, 95,\n      96, 98, 97, 99, 100
↳ ')
>>> mesh = MeshElement(eltype='C3D20', data=[[element1.label, element1]])
```

The class raises a *ElementTypeMismatchError* if we try to add an element with a different eltype:

```
>>> element2 = Element(eltype='CPS4R', datalines='1, 1, 2, 102, 101')
>>> mesh[element2.label] = element2
Traceback (most recent call last):
...
inpRWEErrors.ElementTypeMismatchError: ERROR! An element of type CPS4R cannot be_
↳ added to a MeshElement with eltype C3D20
```

**Parameters**

- **label** (*int*) – The element label.
- **element** (*Element*) – The *Element* to add to the *MeshElement*.

**Raises**

*ElementTypeMismatchError* –

**class** mesh.TotalMesh(\_data=None)

Bases: *csid*

*TotalMesh* is a parent class, and not meant to be instantiated directly. Users should instead work with one of the sub-classes, which are *TotalNodeMesh* and *TotalElementMesh*.

Users should not need to created instances of this class or the child classes directly. In most cases, they can rely on *inpKeywordSequence* to create and manage these instances.

Every *inpKeywordSequence* contains attributes *\_nd* and *\_ed*, which are *TotalNodeMesh* and *TotalElementMesh* instances, respectively. They provide a convenient method to access all the mesh and element data in an input file, without needing to navigate through multiple keyword blocks. These instances of the top-level *inpKeywordSequence* (*keywords*) will be mapped to *nd* and *ed*.

Any operations on items in this class will be propagated back to the data of the appropriate node and element *inpKeyword* blocks.

Items should not be directly deleted from a *TotalMesh*, in most cases. The user should instead use a combination of *findItemReferences()* along with *deleteItemReferences()* to delete these entities, as it will also find and delete all references to those entities throughout the input file.

A function for replacing item references still needs to be written.

Items should not be added to this class directly. Rather, they should be added to *data* of a \*NODE or \*ELEMENT keyword block and added to this class via *updateInp()*.

**`__init__`**(*\_data=None*)

A *TotalMesh* instance holds all the nodes or elements for the children of an *inpKeywordSequence*. This class is meant to be initialized with *\_data* = None. Data should be added to it via *update()* using a *Mesh* instance.

**Parameters**

**`_data`** – Can be any construct used to populate a *dict*. Should not be used in most cases, as the data should instead be populated through a *Mesh*, and then calling *update()* on that *Mesh* instance.

**subMeshes**

Will hold a shared memory reference to every *Mesh* or *TotalMesh* instance which is a child of this object. This is thus a link to the *data* attributes of the appropriate keyword blocks.

**Type**

list

**`removeEmptySubMeshes()`**

Deletes any *subMeshes* which are now empty. This will also delete the *Mesh* instance.

**`renameKeys`**(*mapping*)

Renames all keys in the *TotalMesh* instance (d) as specified by mapping.

d and each *subMesh* will be cleared and repopulated with the renamed keys to preserve the original order. Thus, this can be an expensive operation, so try to provide the entirety of the information to rename and call this function only once.

**`update`**(*other*)

Adds the data from *other* to the *TotalMesh* instance (d), and tracks *other* in *subMeshes*.

**Parameters**

**`other`** (*Mesh*, *TotalMesh*) – A *Mesh* or *TotalMesh* instance whose items will be included in

**`__delitem__`**(*key*)

Deletes key and value in self and in the subMesh.

**Parameters**

**`key`** – A valid hashable key. Should be an integer in most cases, as the node and element labels will be integers.

**`__repr__`**()

Uses the default *\_\_repr\_\_* method from *dict*.

**`__setitem__`**()

Items should not be added to the *TotalMesh* instance. Add them instead to the appropriate *Mesh* instance and then *update()* *TotalMesh* with *Mesh*.

**`__str__`**()

Uses the default *\_\_str\_\_* method from *dict*.

**class** mesh.*TotalNodeMesh*(*\_data=None*)

Bases: *TotalMesh*

A *TotalNodeMesh* enhances *TotalMesh* with some additional functions specific to working with Abaqus nodes.



**setNodesConnectedElements(*nd*)**

This function will set the *elements* attribute of each node in the *TotalMesh* instance for each node found in *nd*.

It should be passed a *Mesh* instance which maps the node labels to the elements connected to those nodes. In most cases, this special *Mesh* instance should be created by *\_parseDataElement()* and it will be available as the *\_nd* attribute of a \*ELEMENT keyword block and as the 'nd\_ed' entry of *\_inpItemsToUpdate*.

**Parameters**

**nd** (*Mesh*) – The *Mesh* instance which contains the node and element label mapping.

**updateNodesConnectedElements(*ed*)**

Calls *updateElements()* on all *Node* instances.

**Parameters**

**ed** (*TotalMesh*) – This should be the *TotalMesh* instance holding the elements.

**Returns**

A dictionary with the node labels as the key, and the return value of *updateElements()* as the value.

**Return type**

dict

**\_\_repr\_\_()**

Uses the default *\_\_repr\_\_* method from dict.

**\_\_str\_\_()**

Uses the default *\_\_str\_\_* method from dict.

**class mesh.TotalElementMesh(\_data=None)**

Bases: *TotalMesh*

A *TotalElementMesh* is merely a subclass of *TotalMesh* with no additional behavior.

**\_\_repr\_\_()**

Uses the default *\_\_repr\_\_* method from dict.

**\_\_str\_\_()**

Uses the default *\_\_str\_\_* method from dict.



## 27.1 Module Contents

This module contains miscellaneous functions that can be used outside of *inpRW*.

`misc_functions.cfc(obj)`

Short for Check For Comment. Checks if *obj* is an Abaqus comment (i.e. begins with `***`). This function will remove spaces and tabs before performing the check.

**Parameters**

**obj** (*str*) – The string to check.

**Returns**

True if `obj.strip(' ')[:2] == '***'`, else False.

**Return type**

*bool*

For example, as long as `***` is at the start of the string, `cfc` will return True:

```
>>> from misc_functions import cfc
>>> cfc('***Node')
True
```

`cfc` also returns True if there are leading spaces and/or tabs:

```
>>> cfc(' ***Node')
True
>>> cfc('  ***Node')
True
```

And of course, it returns False if the item does not lead with `***`:

```
>>> cfc('*Node')
False
```

`misc_functions.chunks(obj, n)`

Splits *obj* into slices of *n* size.

This could be used to split up a large *iterable* for parallel processing.

**Parameters**

**obj** – The object to be sliced into smaller pieces. This must be an *iterable*. **n** (int): The number of objects per slice.

**Yields**

*generator iterator* – The type of the output will be the type of *obj*.

Note that in the following example, converting output and *obj* to lists is for illustrative and testing purposes only.

Example usage:

```
>>> from misc_functions import chunks
>>> obj = list(range(16))
>>> obj
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
>>> list(chunks(obj, n=4))
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]]
```

`misc_functions.flatten(iterable)`

Creates a completely flattened *generator* from a nested sequence.

This function does not care the size of each item in the iterable, that they are consistently sized, or how deep the nesting goes, it will create a one-level deep *generator* from whatever is passed in.

Please note that passing a mapping type to this function will yield only the keys, and that unordered types like sets will be flattened, but the items of the set will not be in any particular order with respect to each other.

**Parameters**

**iterable** – Any *iterable* object. *Strings* will not be processed, as they are naturally flattened.

**Yields**

*generator iterator*

Note that in the following example, converting the output to a list is for illustrative and testing purposes only.

Example usage:

```
>>> from misc_functions import flatten
>>> l = [1, 'abc', [2, 3, [4, 5, (6, 7, 8)]]]
>>> list(flatten(l))
[1, 'abc', 2, 3, 4, 5, 6, 7, 8]
```

`misc_functions.makeDataList(iterable, numPerLine=16)`

Returns a list of lists, with *numPerLine* items per line and then the remainder.

Example usage:

```
>>> from misc_functions import makeDataList
>>> data = list(range(12))
>>> data
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> makeDataList(data, numPerLine=12)
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]]
```

If the number of items in *data* is not evenly divisible by *numPerLine*, this will return as many lists of len *numPerLine* as possible, and then the remaining items in a shorter list. Example:

```
>>> makeDataList(data, numPerLine=8)
[[0, 1, 2, 3, 4, 5, 6, 7], [8, 9, 10, 11]]
```

**Parameters**

- **iterable** (*list*) – A flat iterable.

- **numPerLine** (*int*) – The number of items desired per dataline. Defaults to 16.

#### Returns

A list of lists with numPerLine items on all indices except the last, which will have the remainder.

#### Return type

*list*

`misc_functions.nestedSort(iterable, reverse=False, _level=None)`

Sorts a sequence of sequences on multiple levels.

It starts with the least important field (furthest to the right) and ends with the most important. This function uses `sorted()` with a custom key function to perform the sorting operation. This will return a new iterable.

This was written to sort *positionls* representing the keyword, suboptions, and data position indices. For example, the item at the path '`self.keywords[1].suboptions[0].suboptions[0].data[0][1]`' would have the *positionl* `[1, [0,0], [0,1]]`.

Thus, when sorting keyword *positionls*, `reverse = False` will sort the items in top-down order, while `reverse = True` will sort them in bottom-up order.

#### Parameters

- **iterable** – Any iterable type item with a nested data structure that needs to be sorted.
- **reverse** (*bool*) – If True, iterable will be sorted in reverse (bottom-up) order. Defaults to False.
- **\_level** (*int*) – Allows the user to specify the level to which to sort. Overrides the default value for level, which is `len(iterable[0])`.

#### Returns

A new list with all the items of iterable in the specified order.

#### Return type

*list*

#### Examples

```
>>> from misc_functions import nestedSort
>>> iterable = [[0,[1,0],0], [0,[0,0],0], [1,[1],0], [0,[1,0],1], [1,[0,0],0]]
>>> nestedSort(iterable)
[[0, [0, 0], 0], [0, [1, 0], 0], [0, [1, 0], 1], [1, [0, 0], 0], [1, [1], 0]]
```

And then sorting the original list in reverse order:

```
>>> nestedSort(iterable, reverse=True)
[[1, [1], 0], [1, [0, 0], 0], [0, [1, 0], 1], [0, [1, 0], 0], [0, [0, 0], 0]]
```

Thus, all the right-most items will be sorted first. That order will be maintained while the level one to the left is sorted, until the finish.

`misc_functions.normalize(v)`

This function will return a unit vector in the same direction. *v* should be a vector.

Example:

```
>>> from misc_functions import normalize
>>> normalize([10, 0, 0])
(1.0, 0.0, 0.0)
```

This also works with `Decimal` numbers:

```
>>> from decimal import Decimal as D
>>> normalize([D('15'), D('2'), D('1')])
array([Decimal('0.9890707100936805070769772310'),
       Decimal('0.1318760946791574009435969641'),
       Decimal('0.06593804733957870047179848207')], dtype=object)
```

`misc_functions.ranges(iterable)`

Yields a list of ranges of *iterable*.

This function will determine the ranges needed to represent a sequence of integers.

For example:

```
>>> from misc_functions import ranges
>>> iterable = [1,2,4,5,6,10]
>>> list(ranges(iterable))
[(1, 2), (4, 6), (10, 10)]
```

*iterable* will first be sorted before the functions searches for groups of integers. A gap between integers greater than 1 will start a new group. Duplicate integers are ignored.

This function is used by `_consolidateDofXtoYDataLines()`

**Parameters**

**iterable** – A sequence of integers.

**Yields**

generator iterator

`misc_functions.rsl(obj)`

Lowers the case of *obj*, and then removes all spaces. Name is short for “remove spaces and lowercase”.

`rsl()` is used throughout *inpRW* to provide case- and space-insensitivity on string comparisons.

**Parameters**

**obj** (*str*) – If *obj* is not a *str*, this function will return `None`.

**Returns**

*obj*.lower().replace(' ', ''). `None`: If *obj* is not a string, this function will return `None`.

**Return type**

*str*

Example usage:

```
>>> from misc_functions import rsl
>>> rsl(' This is a test STRING')
'thisisteststring'
```

`misc_functions.rstl(obj)`

Lowers the case of *obj*, and then removes all spaces and tabs. Name is short for “remove spaces, tabs and lowercase”.

`rstl()` can be used throughout *inpRW* to provide case- and white-space-insensitivity on string comparisons. It is slower than `rsl()`, so it will only be used if there are tabs in the input file.

#### Parameters

**obj** (*str*) – If *obj* is not a *str*, this function will return None.

#### Returns

*obj.lower().replace(' ', '').replace(' ', '')*. None: If *obj* is not a string, this function will return None.

#### Return type

*str*

Example usage:

```
>>> from misc_functions import rstl
>>> rstl(' This is a test STRING')
'thisisteststring'
```

`misc_functions.ssl(obj, ss=True)`

Calls `stripSpace()` on *obj.lower()*. Name is short for “strip spaces and lowercase”.

#### Parameters

- **obj** (*str*) –
- **ss** (*bool*) –

#### Returns

*obj.lower()* without leading and trailing spaces if *ss* == True, else *obj.lower()*.

#### Return type

*str*

Example usage:

```
>>> from misc_functions import ssl
>>> ssl(' Test string')
'test string'
```

And if *ss* = False, *obj* is made lowercase and then returned:

```
>>> ssl(' Test string', ss=False)
'test string'
```

`misc_functions.stripSpace(obj, ss=True)`

Strips spaces from the beginning and end of *obj* if *ss* == True.

#### Parameters

- **obj** (*str*) –
- **ss** (*bool*) –

#### Returns

*obj* without leading and trailing spaces if *ss* == True, else *obj*.

#### Return type

*str*

Example usage:

```
>>> from misc_functions import stripSpace
>>> stripSpace('  Test string')
'Test string'
```

The function does returns *obj* unmodified if *ss* = False:

```
>>> stripSpace('  Test string', ss=False)
'  Test string'
```



## 28.1 Module Contents

This module contains a `None`-like class which will always sort to the minimal value.

**class** `NoneSort.NoneSort`

A simple class that allows a `None`-like object to be less than every other value.

A `NoneSort` instance is not `None`, because `None` cannot be subclassed.

Example usage:

```
>>> from NoneSort import NoneSort
>>> ns = NoneSort()
>>> ns < -599899
True
>>> ns > -599899
False
>>> ns == None
True
>>> ns is None
False
>>> bool(ns)
False
```



## 29.1 Module Contents

**class** `printer.Printer`(*data*)

Bases: `object`

Print things to stdout on one line dynamically

**`__init__`**(*data*)

**`__weakref__`**

list of weak references to the object (if defined)



## 30.1 Module Contents

`repr2.repr2(obj, rmtrailing0=False)`

Produces the string representation of *obj* using `str(obj)`.

This function calls `str(obj)` to produce a string representation of *obj*. Thus, *obj* can be any type, so long as `__str__` is defined for *obj*. `Decimal` objects add a trailing `'.'` if not already in the string representation. Can optionally remove trailing 0 if *rmtrailing0* is True (i.e. `1.0 -> 1.`).

### Parameters

- **obj** – Can be any type. `repr2()` will use *obj*'s built-in `__str__` method to produce the string.
- **rmtrailing0** (*bool*) – Remove trailing 0s from decimal numbers if True. Defaults to False.

### Returns

The string representation for *obj*.

### Return type

`str`

Example usage:

```
>>> from repr2 import repr2
>>> repr2(1.0)
'1.0'
>>> from decimal import Decimal
>>> repr2(Decimal('1'))
'1.'
>>> repr2(Decimal('1.0'))
'1.0'
>>> repr2(1.0, rmtrailing0=True)
'1.'
```



## UPDATE NOTES

### 31.1 2023.10.6

This update corrects problems with sub-input files, variable node elements, leading comments, and it adds more tests.

A summary of the major changes follows. Please note that there will be some breaking changes to the API. I have done my best to document all such changes in these notes, so it is imperative that you read these before updating and trying to run an existing script. I will try to minimize API changes in the future, but the performance and robustness of `inpRW` will trump API stability concerns for the near future.

The major features of this update are the following:

- **Improved handling of all multi-input file jobs (for example, `*INCLUDE`, `*MANIFEST`, `INPUT` parameter of multiple keywords).**
  - Added `_subinps` attribute to `inpKeyword` class; this is used only for `*INCLUDE` and `*MANIFEST` blocks and should contain only `inpRW` instances.
  - `inpRW` formerly created sub-instances of the `inpRW` class, but used the `keywords` attribute of the parent instance for the child.
  - When parsing sub-input files (`*MANIFEST`, `*INCLUDE`), a new `inpRW` instance is created for the sub-file, and that instance is placed in the parent block's `_subinps`, and the sub-`inpRW` instance's `keywords` attribute is added to the parent block's `suboptions` attribute.
- The `_elementTypeDictionary` is more useful and more accurate. Each entry is now a new `elType` class instead of a dictionary. `*inpRW` should now accurately understand all element types except for submodel or user elements. Variable node elements were a particular problem before.
- Removed function `inpKeyword.inpKeyword._getElementNodeNum()` and replaced it with `inpKeyword.findElementNodeNum()`.
- Better handling of leading comments (i.e. comment lines before the first keyword block). Previous releases would have difficulties in some cases.
- Corrected a bug related to parsing `*REBAR LAYER` keywords.
- Additional tests added for `inpRW` functionality.
- Addressed an error when the string 'INF' would be converted to a `Decimal` instance instead of treated as a string.
- Added `__repr__()` and `__str__()` functions to the `inpRW` class so they produce useful information.
- Renamed `install.bat` to `install_to_3DEXPERIENCE.bat` to clarify its purpose.
- Corrected several bugs related to string productions, which affected the accuracy of input file writing.
- Corrected a bug where elements from multiple `*ELEMENT` keyword blocks would override the elements attribute of shared node instances.

- Removed python3\_inpRW.bat and python3\_inpRW.sh from the test suite. test\_all.bat and test\_all.sh instead set the INPRW\_TEST environment variable which point to the desired Python interpreter, and the tests have been rewritten to use INPRW\_TEST.
- Expanded the documentation on *Installation and Usage Instructions*. The different options for installing inpRW should be more clear. There is also a new section called *Using inpRW with Keyword Edit in 3DEXPERIENCE*, which includes a video showing how to install *inpRW* to 3DEXPERIENCE and call *inpRW* from a Keyword Edit script.

Unless otherwise noted, if a term can refer to a module name or a class name, I am using it to refer to the class name. Thus, inpKeyword is short-hand for inpKeyword.inpKeyword in these update notes.

### 31.1.1 Performance Improvements:

N/A

### 31.1.2 Modified signatures:

- *inpKeyword.inpKeyword.\_\_str\_\_()*  
Added *includeSubData=True* parameter

### 31.1.3 Modified Modules:

This is a summary of all the modules which have seen some code change. The details are in later sections of this document.

- *inpRW.\_inpR*
- *inpRW.\_inpW*
- *inpRW*
- *csid*
- *eval2*
- *inpDecimal*
- *inpKeyword*
- *inpRWErrors*
- *mesh*
- *config*
- *config\_re*
- *elType*



### 31.1.4 Modified Classes:

This is a summary of all the classes which have seen some code change. The details are in later sections of this document.

- *inpRW.\_inpR.Read*
- *inpRW.\_inpW.Write*
- *inpRW.inpRW*
- *csid.csid*
- *csid.csiKeyString*
- *eval2.eval2*
- *inpDecimal.inpDecimal*
- *inpKeyword*
- *inpRWEErrors.ElementIncorrectNodeNumError*
- *mesh.Element*
- *mesh.TotalNodeMesh*

### 31.1.5 Modified Functions:

- *inpRW.\_inpR.Read.\_readInclude()*  
Improves handling of reading sub-files of \*INCLUDE. Removes leading and trailing whitespace from sub-file name when opening.
- *inpRW.\_inpR.Read.\_readManifest()*  
Improves handling of reading sub-files of \*MANIFEST.
- *inpRW.\_inpR.Read.\_splitKeywords()*  
More accurately identifies comments prior to the first keyword block by using the pattern from *\_getLeadingCommentsPattern()*.
- *inpRW.\_inpW.Write.writeBlocks()*  
Accounts for changes to sub-input file data storage.
- *inpRW.\_inpW.Write.writeInp()*  
No longer sets *\_firstItem* to False after writing leading comments.
- *inpRW.inpRW.\_\_init\_\_()*  
The outputFolder of a sub-instance of *inpRW* will first try to be read from the *inpName* passed to the sub-instance, otherwise it will use *\_parentINP.outputFolder*. Sub-instances of *inpRW* now use their own *inpKeywordSequence* for *keywords* instead of directly sharing the *\_parentINP's* keywords attribute. The *sub-inpKeywordSequence's* contents will still be propagated to the parent *inpRW* instance internally.
- *inpRW.inpRW.parse()*  
Changes to account for new code to identify leading comments.
- *csid.csid.\_\_str\_\_()*  
String keys and/or values will now include quote symbols around them.
- *csid.csiKeyString.\_\_eq\_\_()*  
Will now automatically convert the *other* key to a *csiKeyString* if it is not already.

- `eval2.eval2()`  
Catch *DecimalInfError* and convert the text 'INF' to an *inpString* instead of an *inpDecimal*.  
*inpDecimal*.
- `inpDecimal.inpDecimal.__new__()`  
Raises a *DecimalInfError* if 'INF' (not case-sensitive) is in *inputStr*. This addresses a niche bug and should not affect most users.
- `inpKeyword.inpKeyword.parseKWData()`  
Improved handling of keywords with subdata (i.e. those with the INPUT parameter).
- `inpKeyword.inpKeyword._formatDataLabelDict()`  
Improved handling of comments for \*ELEMENT keyword blocks, specifically those with element definitions spanning more than 1 dataline.
- `inpKeyword.inpKeyword._parseData()`  
Corrected incorrect function call to `_parseRebarLayer()` with the correct call to `_parseDataRebarLayer()`.
- `inpKeyword.inpKeyword._parseDataElement()`  
Modifications to correct inaccurate handling of multi-line and/or variable node element definitions.
- `inpKeyword.inpKeyword._parseDataRebarLayer()`  
Removed some old code which was causing an error.
- `inpKeyword.inpKeyword._parseInputString()`  
Moved call of `_setMiscInpKeywordAttrs()` from `_parseKWLine()` to this function. Corrects rare problem with important parameters for a keyword block not being on the first keyword line. For example, an error would be raised if the NSET parameter of an \*NSET keyword block was not on the first keyword line.
- `inpKeyword.inpKeyword._parseKWLine()`  
Moved call of `_setMiscInpKeywordAttrs()` to `_parseInputString()` from this function.
- `inpKeyword.inpKeyword.__str__()`  
Added *includeSubData* parameter (default to True). This will not be used by end-users in most cases. Setting this parameter to False allows *inpRW* to omit writing sub data to the main input file. The default behavior is identical to the previous release.
- `inpRWErrors.ElementIncorrectNodeNumError.__str__()`  
New options to account for new *nodeNum* data types (i.e. *set* or *int*).
- `mesh.Element.__init__()`  
Updated to account for changes to *\_elementTypeDictionary*.
- `mesh.Element.checkNumberNodes()`  
Updated to account for changes to *\_elementTypeDictionary*.
- `mesh.Element.__str__()`  
Corrected a problem with string production for multi-line elements.
- `mesh.TotalNodeMesh.setNodesConnectedElements()`  
Corrected a bug which overrode the elements connected to a node if the node was connected to different elements defined in multiple \*ELEMENT keyword blocks.

### 31.1.6 Modified Attributes:

- `config._elementTypeDictionary`

The value for each element is now the new `elType` class instead of a `csid`. `inpRW` has been updated to account for this change, but any user scripts which directly accessed this dictionary will need to be updated to account for the new structure. This dictionary should include more information about each element type, (description, valid solver, and number of nodes), and it identifies variable node elements (they use `numNodesSet` instead of `numNodesSet`).

### 31.1.7 New Functions:

- `inpRW.inpRW._getLeadingCommentsPattern()`

Finds the location where the first keyword block begins and returns a regular expression pattern to identify that location.

- `inpRW.inpRW.__repr__()`

Produces a useful string representation of an `inpRW` instance.

- `inpRW.inpRW.__str__()`

Produces a string which helps identify the input file which corresponds to this `inpRW.inpRW` instance.

- `inpKeyword.findElementNodeNum()`

Replaces and simplifies `inpKeyword.inpKeyword._getElementNodeNum()`. With the exception of user elements, sub-structure elements, and variable number elements, all Abaqus elements have a static number of nodes which can define them. Thus, this function simply performs a dictionary lookup in most cases. For variable node elements, this function looks up the valid number of nodes to define the element. Then, it keeps reading data lines until it finds a line which does not end with a comma. If the number of nodes is in the set of valid node numbers for the element type, this function returns an integer. If the number of nodes is not in the set, an `ElementIncorrectNodeNumError` is raised. For undefined element type labels, this function will read datalines until it finds one which does not end with a comma. It will return the number of nodes it thinks the element has. For accurate results with user or sub-structure elements, the user should add an entry to `inp._elementTypeDictionary` before calling `inp.parse`.

### 31.1.8 New Attributes:

- `inpKeyword.inpKeyword._subinps`

Contains sub `inpRW` instances for \*INCLUDE and \*MANIFEST blocks.

- `config_re.re25`

Used to find leading comments prior to first keyword.

- `config_re.re26`

Used to check if an element definition has been completed.

### 31.1.9 New Modules:

- ***elType***  
Provides the *elType* class.
- ***paramTypes***  
Internal only module, not yet shipped with *inpRW*. The handling of \*PARAMETER definition and references needs to be improved, and this module is the foundation of that future work.

### 31.1.10 New Classes:

- ***elType.elType***  
Stores information about an Abaqus element type. Used for each entry in *\_elementTypeDictionary* instead of sub *csid* instances. Includes the element description from the Abaqus documentation, the valid solver(s), and the number of nodes needed to define the element.
- ***inpRWErrors.DecimalInfError***  
This exception is raised by *inpDecimal.inpDecimal.\_\_new\_\_()* if the input string is 'INF'. It is meant to handle a very niche error.

### 31.1.11 Removed:

- ***inpKeyword.inpKeyword.\_getElementNodeNum()***  
Replaced with *inpKeyword.findElementNodeNum()*.
- Removed *python3\_inpRW.bat* and *python3\_inpRW.sh* from the test suite.

### 31.1.12 Automated Tests:

Added automated tests to the following items:

- ***elType* module:**  
all class functions
- ***inpKeyword* module:**
  - *findElementNodeNum()*
  - *printKWandSuboptions()*
  - *writeKWandSuboptions()*
  - *\_formatDataOutput()*
  - *\_formatDataLabelDict()*
- ***test\_Example\_Scenarios*** This test runs the example script from *Example Scenarios* page of the documentation.

### 31.1.13 Misc:

N/A



## LEGAL NOTICES AND USAGE DISCLAIMER

### 32.1 inpRW Usage Disclaimer

BY DOWNLOADING THIS FILE (“DOWNLOAD”) YOU AGREE TO THE FOLLOWING TERMS:

THIS DOWNLOAD IS MADE AVAILABLE ON AN “AS IS” BASIS WITHOUT WARRANTY OF ANY KIND, WHETHER EXPRESS OR IMPLIED, ORAL OR WRITTEN, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE OR NON-INFRINGEMENT.

DASSAULT SYSTEMES SIMULIA CORP., ANY OF IT AFFILIATES (COLLECTIVELY “DS.”), AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR DIRECT, INDIRECT, INCIDENTAL, CONSEQUENTIAL OR PUNITIVE DAMAGES, INCLUDING WITHOUT LIMITATION CLAIMS FOR LOST PROFITS, BUSINESS INTERRUPTION AND LOSS OF DATA, THAT IN ANY WAY RELATE TO THIS DOWNLOAD, WHETHER OR NOT DS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES AND NOTWITHSTANDING THE FAILURE OF THE ESSENTIAL PURPOSE OF ANY REMEDY.

YOUR USE OF THIS DOWNLOAD SHALL BE AT YOUR SOLE RISK. NO SUPPORT OF ANY KIND OF THE DOWNLOAD IS PROVIDED BY DS.

### 32.2 Legal Notices

Legal Notices





## PYTHON MODULE INDEX

### C

centroid, 69  
config, 71  
config\_re, 75  
csid, 77

### e

elType, 91  
eval2, 89

### i

inpDecimal, 95  
inpInt, 99  
inpKeyword, 103  
inpKeywordHelper, 127  
inpKeywordSequence, 129  
inpRW, 35  
inpRW.\_importedModules, 68  
inpRW.\_inpCustom, 68  
inpRW.\_inpFind, 45  
inpRW.\_inpFindRefs, 52  
inpRW.\_inpMod, 60  
inpRW.\_inpR, 41  
inpRW.\_inpW, 45  
inpRWErrors, 133  
inpString, 137

### m

mesh, 141  
misc\_functions, 157

### n

NoneSort, 163

### p

printer, 165

### r

repr2, 167



## Symbols

- `_EndKWs` (*in module config*), 72
- `_EoFKWs` (*in module config*), 72
- `_ParamInInp` (*inpKeyword.inpKeyword attribute*), 109
- `_ParamInInp` (*inpRW.inpRW attribute*), 38
- `_ParamInInp` (*mesh.Element attribute*), 147
- `_ParamInInp` (*mesh.Node attribute*), 143
- `__contains__` (*csid.csid method*), 80
- `__delitem__` (*csid.csid method*), 81
- `__delitem__` (*mesh.TotalMesh method*), 154
- `__eq__` (*csid.csiKeyDecimal method*), 86
- `__eq__` (*csid.csiKeyString method*), 83
- `__getattr__` (*elType.elType method*), 92
- `__getattr__` (*inpKeyword.inpKeyword method*), 124
- `__getitem__` (*csid.csid method*), 81
- `__getnewargs__` (*inpInt.inpInt method*), 101
- `__getnewargs__` (*inpString.inpString method*), 139
- `__hash__` (*csid.csiKeyDecimal method*), 85
- `__hash__` (*csid.csiKeyString method*), 83
- `__init__` (*csid.csiKeyDecimal method*), 85
- `__init__` (*csid.csiKeyString method*), 82
- `__init__` (*csid.csid method*), 77
- `__init__` (*elType.elType method*), 91
- `__init__` (*inpDecimal.inpDecimal method*), 95
- `__init__` (*inpInt.inpInt method*), 99
- `__init__` (*inpKeyword.inpKeyword method*), 107
- `__init__` (*inpKeywordSequence.inpKeywordSequence method*), 129
- `__init__` (*inpRW.inpRW method*), 35
- `__init__` (*inpRWErrors.ElementIncorrectNodeNumError method*), 134
- `__init__` (*inpRWErrors.ElementTypeMismatchError method*), 135
- `__init__` (*inpRWErrors.inpRWErrors method*), 133
- `__init__` (*inpString.inpString method*), 137
- `__init__` (*mesh.Element method*), 144
- `__init__` (*mesh.Mesh method*), 151
- `__init__` (*mesh.MeshElement method*), 152
- `__init__` (*mesh.Node method*), 141
- `__init__` (*mesh.TotalMesh method*), 154
- `__init__` (*printer.Printer method*), 165
- `__members__` (*inpRW.inpRW attribute*), 38
- `__methods__` (*inpRW.inpRW attribute*), 38
- `__new__` (*csid.csiKeyDecimal static method*), 84
- `__new__` (*inpDecimal.inpDecimal static method*), 95
- `__new__` (*inpInt.inpInt static method*), 99
- `__new__` (*inpString.inpString static method*), 137
- `__reduce__` (*inpDecimal.inpDecimal method*), 97
- `__repr__` (*csid.csiKeyDecimal method*), 86
- `__repr__` (*csid.csiKeyString method*), 84
- `__repr__` (*csid.csid method*), 82
- `__repr__` (*elType.elType method*), 93
- `__repr__` (*inpDecimal.inpDecimal method*), 97
- `__repr__` (*inpInt.inpInt method*), 100
- `__repr__` (*inpKeyword.inpKeyword method*), 125
- `__repr__` (*inpRW.inpRW method*), 36
- `__repr__` (*inpString.inpString method*), 139
- `__repr__` (*mesh.Element method*), 150
- `__repr__` (*mesh.Mesh method*), 151
- `__repr__` (*mesh.Node method*), 144
- `__repr__` (*mesh.TotalElementMesh method*), 155
- `__repr__` (*mesh.TotalMesh method*), 154
- `__repr__` (*mesh.TotalNodeMesh method*), 155
- `__setitem__` (*csid.csid method*), 82
- `__setitem__` (*mesh.MeshElement method*), 153
- `__setitem__` (*mesh.TotalMesh method*), 154
- `__str__` (*csid.csiKeyDecimal method*), 86
- `__str__` (*csid.csiKeyString method*), 84
- `__str__` (*csid.csid method*), 82
- `__str__` (*elType.elType method*), 93
- `__str__` (*inpDecimal.inpDecimal method*), 97
- `__str__` (*inpInt.inpInt method*), 100
- `__str__` (*inpKeyword.inpKeyword method*), 125
- `__str__` (*inpRW.inpRW method*), 36
- `__str__` (*inpRWErrors.DecimalInfError method*), 134
- `__str__` (*inpRWErrors.ElementIncorrectNodeNumError method*), 135
- `__str__` (*inpRWErrors.ElementTypeMismatchError method*), 135
- `__str__` (*inpRWErrors.inpRWErrors method*), 133
- `__str__` (*inpString.inpString method*), 139
- `__str__` (*mesh.Element method*), 150
- `__str__` (*mesh.Mesh method*), 151

\_\_str\_\_() (mesh.Node method), 144  
 \_\_str\_\_() (mesh.TotalElementMesh method), 155  
 \_\_str\_\_() (mesh.TotalMesh method), 154  
 \_\_str\_\_() (mesh.TotalNodeMesh method), 155  
 \_\_weakref\_\_ (csid.csiKeyDecimal attribute), 86  
 \_\_weakref\_\_ (csid.csiKeyString attribute), 84  
 \_\_weakref\_\_ (csid.csid attribute), 82  
 \_\_weakref\_\_ (elType.elType attribute), 94  
 \_\_weakref\_\_ (inpDecimal.inpDecimal attribute), 97  
 \_\_weakref\_\_ (inpKeyword.inpKeyword attribute), 124  
 \_\_weakref\_\_ (inpRWErrors.inpRWErrors attribute), 133  
 \_\_weakref\_\_ (inpString.inpString attribute), 139  
 \_\_weakref\_\_ (mesh.Element attribute), 151  
 \_\_weakref\_\_ (mesh.Node attribute), 144  
 \_\_weakref\_\_ (printer.Printer attribute), 165  
 \_addSpace (inpKeyword.inpKeyword attribute), 109  
 \_addSpace (inpRW.inpRW attribute), 39  
 \_allKwNames (in module config), 72  
 \_appendDatelineKeywordsOP (in module config), 71  
 \_bad\_kwblock (inpRW.inpRW attribute), 39  
 \_baseStep (inpKeyword.inpKeyword attribute), 109  
 \_checknumNodes (mesh.Element attribute), 147  
 \_cloneKW() (inpKeyword.inpKeyword method), 118  
 \_consolidateDofXtoYDatelines() (inpRW.\_inpMod.Mod method), 66  
 \_convertSystem() (inpRW.\_inpMod.Mod method), 67  
 \_couplingSurfNames (inpRW.inpRW attribute), 39  
 \_createSubKW() (inpRW.\_inpR.Read method), 43  
 \_curBaseStep (inpRW.inpRW attribute), 39  
 \_data (inpKeyword.inpKeyword attribute), 110  
 \_dataKWs (in module config), 72  
 \_dataParsed (inpKeyword.inpKeyword attribute), 111  
 \_debug (inpKeyword.inpKeyword attribute), 109  
 \_debug (inpRW.inpRW attribute), 39  
 \_delayedPlaceBlocks (inpKeywordSequence.inpKeywordSequence attribute), 130  
 \_delayedPlaceBlocks (inpRW.inpRW attribute), 39  
 \_distCoupElsetNames (inpRW.inpRW attribute), 39  
 \_dofXtoYDatelineKeywordsOP (in module config), 71  
 \_ed (inpKeywordSequence.inpKeywordSequence attribute), 130  
 \_elementTypeDictionary (in module config), 71  
 \_endSubKW() (inpRW.\_inpR.Read method), 43  
 \_evalDecimal() (inpDecimal.inpDecimal method), 96  
 \_evalString() (inpString.inpString method), 138  
 \_findActiveSystem() (inpRW.\_inpFind.Find method), 48  
 \_findData() (inpRW.\_inpFind.Find method), 50  
 \_findIncludeFileNames() (inpRW.\_inpFind.Find method), 51  
 \_findParam() (inpRW.\_inpFind.Find method), 51  
 \_firstItem (inpKeyword.inpKeyword attribute), 109  
 \_firstItem (inpRW.inpRW attribute), 39  
 \_formatDataLabelDict() (inpKeyword.inpKeyword method), 119  
 \_formatDataOutput() (inpKeyword.inpKeyword method), 119  
 \_formatExp (inpDecimal.inpDecimal attribute), 96  
 \_formatStr (inpDecimal.inpDecimal attribute), 96  
 \_formatStr (inpInt.inpInt attribute), 99  
 \_formatStr (inpString.inpString attribute), 138  
 \_generalSteps (in module config), 72  
 \_getLastBlockPath() (inpRW.\_inpFind.Find method), 52  
 \_getLeadingCommentsPattern() (inpRW.\_inpR.Read method), 43  
 \_grandchildBlocks (inpKeywordSequence.inpKeywordSequence attribute), 130  
 \_groupKeywordBlocks() (inpRW.\_inpR.Read method), 44  
 \_handleComment() (inpKeyword.inpKeyword method), 120  
 \_handleCommentSub() (inpKeyword.inpKeyword method), 120  
 \_inpItemsToUpdate (inpKeyword.inpKeyword attribute), 110  
 \_inpText (inpRW.inpRW attribute), 39  
 \_insertComments() (in module inpKeyword), 106  
 \_isolateNodesToKeep() (inpRW.\_inpMod.Mod method), 67  
 \_joinPS (inpKeyword.inpKeyword attribute), 109  
 \_joinPS (inpRW.inpRW attribute), 39  
 \_joinS (mesh.Element attribute), 148  
 \_joinS (mesh.Node attribute), 143  
 \_keywordsDelayPlacingData (in module config), 72  
 \_kinCoupNsetNames (inpRW.inpRW attribute), 39  
 \_kw (inpRW.inpRW attribute), 39  
 \_kwsumFile (inpRW.inpRW attribute), 39  
 \_kwsumName (inpRW.inpRW attribute), 39  
 \_lbp (inpRW.inpRW attribute), 39  
 \_leadingComments (inpRW.inpRW attribute), 40  
 \_leadstr (inpKeyword.inpKeyword attribute), 109  
 \_lineEnd (mesh.Element attribute), 148  
 \_manBaseStep (inpRW.inpRW attribute), 40  
 \_maxParsingLoops (in module config), 73  
 \_mergeDatelineKeywordsOP (in module config), 71  
 \_mergeElementData() (mesh.Node method), 144  
 \_namedRefs (inpKeyword.inpKeyword attribute), 110  
 \_namedRefs (inpKeywordSequence.inpKeywordSequence attribute), 130  
 \_nd (inpKeyword.inpKeyword attribute), 110  
 \_nd (inpKeywordSequence.inpKeywordSequence attribute), 130  
 \_nl (inpKeyword.inpKeyword attribute), 109

\_nl (*inpRW.inpRW attribute*), 40  
 \_nl (*mesh.Element attribute*), 148  
 \_npl1 (*mesh.Element attribute*), 148  
 \_numCpus (*inpRW.inpRW attribute*), 40  
 \_opKeywords (*in module config*), 72  
 \_openEncoding (*in module config*), 71  
 \_outstr() (*inpDecimal.inpDecimal method*), 96  
 \_outstr() (*inpInt.inpInt method*), 100  
 \_outstr() (*inpString.inpString method*), 138  
 \_outstr\_lead0() (*inpInt.inpInt method*), 100  
 \_parentINP (*inpRW.inpRW attribute*), 40  
 \_parentblock (*inpRW.inpRW attribute*), 40  
 \_parentkws (*inpRW.inpRW attribute*), 40  
 \_parseData() (*inpKeyword.inpKeyword method*), 121  
 \_parseDataElement() (*inpKeyword.inpKeyword method*), 121  
 \_parseDataGeneral() (*inpKeyword.inpKeyword method*), 121  
 \_parseDataHeading() (*inpKeyword.inpKeyword method*), 121  
 \_parseDataLine() (*mesh.Element method*), 149  
 \_parseDataLine() (*mesh.Node method*), 143  
 \_parseDataLines() (*mesh.Element method*), 149  
 \_parseDataNode() (*inpKeyword.inpKeyword method*), 122  
 \_parseDataParameter() (*inpKeyword.inpKeyword method*), 122  
 \_parseDataRebarLayer() (*inpKeyword.inpKeyword method*), 122  
 \_parseDataSet() (*inpKeyword.inpKeyword method*), 122  
 \_parseDataSurface() (*inpKeyword.inpKeyword method*), 123  
 \_parseInputString() (*inpKeyword.inpKeyword method*), 123  
 \_parseKWLine() (*inpKeyword.inpKeyword method*), 123  
 \_parseSubData() (*inpKeyword.inpKeyword method*), 123  
 \_pd (*inpKeywordSequence.inpKeywordSequence attribute*), 130  
 \_perturbationSteps (*in module config*), 72  
 \_placeInpItemsToUpdate() (*inpKeywordSequence.inpKeywordSequence method*), 131  
 \_populateRebarLayerNamedRefs() (*inpKeyword.inpKeyword method*), 124  
 \_processDataLinesInput() (*mesh.Element method*), 150  
 \_readInclude() (*inpRW.\_inpR.Read method*), 44  
 \_readManifest() (*inpRW.\_inpR.Read method*), 44  
 \_removeSystem() (*inpRW.\_inpMod.Mod method*), 67  
 \_replaceStarParameter() (*inpRW.\_inpMod.Mod method*), 68  
 \_setMiscInpKeywordAttrs() (*inpKeyword.inpKeyword method*), 124  
 \_slash (*in module config*), 71  
 \_splitKeywords() (*inpRW.\_inpR.Read method*), 44  
 \_subBlockKWs (*in module config*), 72  
 \_subP (*inpRW.inpRW attribute*), 40  
 \_subParam() (*inpRW.\_inpR.Read method*), 44  
 \_subdata (*inpKeyword.inpKeyword attribute*), 110  
 \_subinps (*inpKeyword.inpKeyword attribute*), 109  
 \_subkwin (*inpRW.inpRW attribute*), 40  
 \_supFilePath (*in module config*), 71  
 \_tree (*inpRW.inpRW attribute*), 40  
 \_value (*inpString.inpString attribute*), 137  
 \_yieldInpRWItemsToUpdate() (*inpKeyword.inpKeyword method*), 124  
 3DExperience, 33

## A

Abaqus, 33  
 append() (*inpKeywordSequence.inpKeywordSequence method*), 130  
 averageVertices() (*in module centroid*), 69

## B

block, 33

## C

calculateCentroid() (*inpRW.\_inpMod.Mod method*), 60  
 centroid  
   module, 69  
 cfc() (*in module misc\_functions*), 157  
 checkNumberNodes() (*mesh.Element method*), 148  
 chunks() (*in module misc\_functions*), 157  
 comments (*inpKeyword.inpKeyword attribute*), 111  
 config  
   module, 71  
 config\_re  
   module, 75  
 convertOPnewToOPmod() (*inpRW.\_inpMod.Mod method*), 60  
 createManifest() (*inpRW.\_inpMod.Mod method*), 61  
 createParamDictFromString() (*in module inpKeyword*), 103  
 createPathFromSequence() (*inpRW.\_inpR.Read method*), 41  
 csid  
   module, 77  
 csid (*class in csid*), 77  
 csiKeyDecimal (*class in csid*), 84  
 csiKeyString (*class in csid*), 82  
 Custom (*class in inpRW.\_inpCustom*), 68

## D

data, 33  
 data (*inpKeyword.inpKeyword* attribute), 110  
 data (*mesh.Element* attribute), 148  
 data (*mesh.Node* attribute), 143  
 data item, 33  
 dataline, 33  
 DecimalInfError, 134  
 delayParsingDataKws (*inpKeyword.inpKeyword* attribute), 109  
 delayParsingDataKws (*inpRW.inpRW* attribute), 37  
 deleteItemReferences() (*inpRW.\_inpMod.Mod* method), 62  
 deleteKeyword() (*inpRW.\_inpMod.Mod* method), 63  
 description (*elType.elType* attribute), 92

## E

ed (*inpRW.inpRW* attribute), 37  
 Element (class in *mesh*), 144  
 ElementIncorrectNodeNumError, 134  
 elements (*mesh.Node* attribute), 143  
 ElementTypeMismatchError, 135  
 elType  
   module, 91  
 elType (class in *elType*), 91  
 eltype (*mesh.Element* attribute), 147  
 eltype (*mesh.MeshElement* attribute), 152  
 eval2  
   module, 89  
 eval2() (in module *eval2*), 89  
 extend() (*inpKeywordSequence.inpKeywordSequence* method), 131

## F

fastElementParsing (*inpKeyword.inpKeyword* attribute), 109  
 fastElementParsing (*inpRW.inpRW* attribute), 37  
 Find (class in *inpRW.\_inpFind*), 45  
 findAdaptivemeshcontrolsRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 52  
 findClearanceRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 53  
 findCloseNodes() (*inpRW.\_inpFind.Find* method), 48  
 findConnectorbehaviorRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 53  
 findContactclearanceRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 53  
 findContactinitializationdataRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 53  
 findContactpairRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 54  
 findDistributionRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 54

findDistributiontableRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 54  
 findElementNodeNum() (in module *inpKeyword*), 103  
 findElementprogressiveactivationRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 54  
 findElementRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 55  
 findEnrichmentRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 55  
 findFastenerpropertyRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 55  
 findFieldmappercontrolsRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 55  
 findFluidbehaviorRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 56  
 findGasketbehaviorRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 56  
 findImpedancepropertyRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 56  
 findIncidentwaveinteractionpropertyRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 56  
 findIncidentwavepropertyRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 56  
 findIntegratedoutputsectionRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 57  
 findItemReferences() (*inpRW.\_inpFind.Find* method), 45  
 findKeyword() (*inpRW.\_inpFind.Find* method), 46  
 findKeywordsIgnoreParam() (*inpRW.\_inpFind.Find* method), 47  
 findNodeRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 57  
 findOrientationRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 57  
 findPeriodicmediaRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 57  
 findRebarlayerRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 58  
 findRebarRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 58  
 FindRefs (class in *inpRW.\_inpFindRefs*), 52  
 findRigidbodyRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 58  
 findSectioncontrolsRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 58  
 findStepLocations() (*inpRW.\_inpFind.Find* method), 48  
 findSubstructureloadcaseRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 59  
 findSurfaceinteractionRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 59  
 findSurfacepropertyRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 59  
 findSurfaceRefs() (*inpRW.\_inpFindRefs.FindRefs* method), 59



- method*), 59  
 findSurfacesmoothingRefs() (*inpRW.\_inpFindRefs.FindRefs method*), 60  
 findTracerparticleRefs() (*inpRW.\_inpFindRefs.FindRefs method*), 60  
 findValidParentKW() (*inpRW.\_inpFind.Find method*), 48  
 flatten() (*in module misc\_functions*), 158  
 formatData() (*inpKeyword.inpKeyword method*), 111  
 formatKeywordLine() (*inpKeyword.inpKeyword method*), 112  
 formatParameterOutput() (*inpKeyword.inpKeyword method*), 113  
 formatStringFromParamDict() (*in module inpKeyword*), 105
- ## G
- generateInpSum() (*inpRW.\_inpW.Write method*), 45  
 get() (*csid.csid method*), 78  
 getLabelsFromSet() (*inpRW.\_inpMod.Mod method*), 64  
 getParentBlock() (*inpRW.\_inpFind.Find method*), 48
- ## I
- includeBlockPaths (*inpRW.inpRW attribute*), 37  
 includeFileNames (*inpRW.inpRW attribute*), 37  
 inpDecimal  
   module, 95  
 inpDecimal (*class in inpDecimal*), 95  
 inpInt  
   module, 99  
 inpInt (*class in inpInt*), 99  
 inpKeyword  
   module, 103  
 inpKeyword (*class in inpKeyword*), 107  
 inpKeywordArgs (*inpRW.inpRW attribute*), 37  
 inpKeywordHelper  
   module, 127  
 inpKeywordHelper() (*in module inpKeywordHelper*), 127  
 inpKeywordInit() (*in module inpKeywordHelper*), 127  
 inpKeywordSequence  
   module, 129  
 inpKeywordSequence (*class in inpKeywordSequence*), 129  
 inpName (*inpRW.inpRW attribute*), 37  
 inpParser, 33  
 inpRW  
   module, 35  
 inpRW (*class in inpRW*), 35  
 inpRW.\_importedModules  
   module, 68  
 inpRW.\_inpCustom  
   module, 68  
 inpRW.\_inpFind  
   module, 45  
 inpRW.\_inpFindRefs  
   module, 52  
 inpRW.\_inpMod  
   module, 60  
 inpRW.\_inpR  
   module, 41  
 inpRW.\_inpW  
   module, 45  
 inpRWErrors, 133  
   module, 133  
 inpString  
   module, 137  
 inpString (*class in inpString*), 137  
 input file, 33  
 inputFolder (*inpKeyword.inpKeyword attribute*), 109  
 inputFolder (*inpRW.inpRW attribute*), 37  
 inputString (*inpKeyword.inpKeyword attribute*), 110  
 insert() (*inpKeywordSequence.inpKeywordSequence method*), 131  
 insertKeyword() (*inpRW.\_inpMod.Mod method*), 64
- ## J
- job, 33  
 jobSuffix (*inpRW.inpRW attribute*), 37
- ## K
- key (*csid.csiKeyString attribute*), 83  
 keyword, 33  
 keyword block, 33  
 Keyword Edit, 33  
 keyword line, 34  
 KeywordNotFoundError, 133  
 keywords (*inpRW.inpRW attribute*), 37  
 ktridd (*inpRW.inpRW attribute*), 37  
 kwg (*inpRW.inpRW attribute*), 37
- ## L
- label (*mesh.Element attribute*), 148  
 label (*mesh.Node attribute*), 143  
 lines (*inpKeyword.inpKeyword attribute*), 111
- ## M
- makeDataList() (*in module misc\_functions*), 158  
 mergeNodes() (*inpRW.\_inpMod.Mod method*), 64  
 mergeSubItems() (*csid.csid method*), 78  
 mergeSuboptionsGlobalData() (*inpKeywordSequence.inpKeywordSequence method*), 131  
 mesh  
   module, 141  
 Mesh (*class in mesh*), 151  
 MeshElement (*class in mesh*), 152

misc\_functions  
  module, 157  
Mod (*class in inpRW.\_inpMod*), 60  
module

  centroid, 69  
  config, 71  
  config\_re, 75  
  csid, 77  
  elType, 91  
  eval2, 89  
  inpDecimal, 95  
  inpInt, 99  
  inpKeyword, 103  
  inpKeywordHelper, 127  
  inpKeywordSequence, 129  
  inpRW, 35  
  inpRW.\_importedModules, 68  
  inpRW.\_inpCustom, 68  
  inpRW.\_inpFind, 45  
  inpRW.\_inpFindRefs, 52  
  inpRW.\_inpMod, 60  
  inpRW.\_inpR, 41  
  inpRW.\_inpW, 45  
  inpRWErrors, 133  
  inpString, 137  
  mesh, 141  
  misc\_functions, 157  
  NoneSort, 163  
  printer, 165  
  repr2, 167

## N

name (*elType.elType attribute*), 92  
name (*inpKeyword.inpKeyword attribute*), 110  
namedRefs (*inpRW.inpRW attribute*), 37  
nd (*inpRW.inpRW attribute*), 37  
nestedSort() (*in module misc\_functions*), 159  
Node (*class in mesh*), 141  
nodesUpdatedConnectivity (*inpRW.inpRW attribute*), 37  
NoneSort  
  module, 163  
NoneSort (*class in NoneSort*), 163  
normalize() (*in module misc\_functions*), 159  
numNodes (*elType.elType attribute*), 92  
numNodes (*mesh.Element attribute*), 147  
numNodesSet (*elType.elType attribute*), 92

## O

organize (*inpRW.inpRW attribute*), 38  
outputFolder (*inpRW.inpRW attribute*), 38

## P

parameter, 34

parameter (*inpKeyword.inpKeyword attribute*), 110  
parent block, 34  
parentBlock (*inpKeywordSequence.inpKeywordSequence attribute*), 129  
parse() (*inpRW.inpRW method*), 36  
parseKWData() (*inpKeyword.inpKeyword method*), 113  
parsePath() (*inpRW.\_inpR.Read method*), 41  
parseSubFiles (*inpKeyword.inpKeyword attribute*), 109  
parseSubFiles (*inpRW.inpRW attribute*), 38  
path (*inpKeyword.inpKeyword attribute*), 111  
pd (*inpKeyword.inpKeyword attribute*), 110  
pd (*inpRW.inpRW attribute*), 38  
pip, 34  
pktridc (*inpRW.inpRW attribute*), 38  
pop() (*csid.csid method*), 79  
positionl, 34  
pre-processor, 34  
preserveSpacing (*inpKeyword.inpKeyword attribute*), 108  
preserveSpacing (*inpRW.inpRW attribute*), 38  
preserveSpacing (*mesh.Element attribute*), 147  
preserveSpacing (*mesh.Node attribute*), 143  
printDocs() (*inpRW.inpRW method*), 36  
printer  
  module, 165  
Printer (*class in printer*), 165  
printKWandSuboptions() (*inpKeyword.inpKeyword method*), 116  
printSubBlocks (*inpKeywordSequence.inpKeywordSequence attribute*), 130

## R

ranges() (*in module misc\_functions*), 160  
re1 (*in module config\_re*), 75  
re10 (*in module config\_re*), 75  
re11 (*in module config\_re*), 75  
re12 (*in module config\_re*), 75  
re13 (*in module config\_re*), 75  
re14 (*in module config\_re*), 75  
re15 (*in module config\_re*), 76  
re16 (*in module config\_re*), 76  
re17 (*in module config\_re*), 76  
re18 (*in module config\_re*), 76  
re19 (*in module config\_re*), 76  
re2 (*in module config\_re*), 75  
re21 (*in module config\_re*), 76  
re22 (*in module config\_re*), 76  
re23 (*in module config\_re*), 76  
re24 (*in module config\_re*), 76  
re25 (*in module config\_re*), 76  
re26 (*in module config\_re*), 76



re3 (in module *config\_re*), 75  
 re4 (in module *config\_re*), 75  
 re5 (in module *config\_re*), 75  
 re7 (in module *config\_re*), 75  
 re8 (in module *config\_re*), 75  
 re9 (in module *config\_re*), 75  
 Read (class in *inpRW.\_inpR*), 41  
 reduceStepSubKWs() (*inpRW.\_inpMod.Mod* method), 65  
 removeEmptySubMeshes() (*mesh.TotalMesh* method), 154  
 removeGenerate() (*inpRW.\_inpMod.Mod* method), 65  
 renameKeys() (*mesh.TotalMesh* method), 154  
 replaceKeyword() (*inpRW.\_inpMod.Mod* method), 66  
 repr2  
     module, 167  
 repr2() (in module *repr2*), 167  
 rmtrailing0 (*inpKeyword.inpKeyword* attribute), 109  
 rmtrailing0 (*inpRW.inpRW* attribute), 38  
 rsl() (in module *misc\_functions*), 160  
 rstl() (in module *misc\_functions*), 160

## S

setConnectedNodes() (*mesh.Element* method), 150  
 setdefault() (*csid.csid* method), 79  
 setNodesConnectedElements()  
     (*mesh.TotalNodeMesh* method), 154  
 solvers (*elType.elType* attribute), 92  
 sortKWs() (*inpRW.\_inpR.Read* method), 42  
 ss (*inpRW.inpRW* attribute), 38  
 ss (*inpString.inpString* attribute), 137  
 ssl() (in module *misc\_functions*), 161  
 step\_paths (*inpRW.inpRW* attribute), 38  
 steps (*inpRW.inpRW* attribute), 38  
 stripSpace() (in module *misc\_functions*), 161  
 subblock, 34  
 subBlockLoop() (*inpRW.\_inpR.Read* method), 42  
 subkeyword, 34  
 subMeshes (*mesh.TotalMesh* attribute), 154  
 suboption, 34  
 suboptions (*inpKeyword.inpKeyword* attribute), 111

## T

TotalElementMesh (class in *mesh*), 155  
 TotalMesh (class in *mesh*), 153  
 TotalNodeMesh (class in *mesh*), 154

## U

update() (*csid.csid* method), 80  
 update() (*mesh.TotalMesh* method), 154  
 updateElements() (*mesh.Node* method), 144  
 updateInp() (*inpRW.inpRW* method), 36  
 updateKTRIDD() (*inpRW.\_inpMod.Mod* method), 66

updateNodesConnectedElements()  
     (*mesh.TotalNodeMesh* method), 155  
 updateObjectsPath() (*inpRW.\_inpR.Read* method), 43  
 updatePKTRICD() (*inpRW.\_inpMod.Mod* method), 66  
 useDecimal (*inpKeyword.inpKeyword* attribute), 109  
 useDecimal (*inpRW.inpRW* attribute), 38  
 useDecimal (*mesh.Element* attribute), 148  
 useDecimal (*mesh.Node* attribute), 143  
 user script, 34

## W

Write (class in *inpRW.\_inpW*), 45  
 writeBlocks() (*inpRW.\_inpW.Write* method), 45  
 writeInp() (*inpRW.\_inpW.Write* method), 45  
 writeKWandSuboptions() (*inpKeyword.inpKeyword* method), 117