PandaCoreData

Sep 26, 2019

Tutorial

So, if you already played games like Factorio, Rimworld or Dwarf Fortress and tried to mod them, you will find that it's absurdly easy to mod those games for simple things like changing the balancing, adding items, changing descriptions and etc. Because the data files of the games are simple raws and not binaries. This library pretty much makes the task of using raws as simple as possible. Raws in our case is plain text files like xml, json, yaml and etc that is commonly used to hold data, for now the library supports *yaml* and *json*.

The library might have modding in mind, however it pretty much be used for any sort of software or game engine that uses python.

Internally the library use dataclasses to handle the data and TinyDB to load them from raws.

CHAPTER 1

Getting Started

First of all we need to install the library so we can use it! So here we go:

1.1 Installation

Python have a lot of options if you want to install a package. Since the library is available in Pypi repository all you need to run most of the time is:

```
pip install panda_core_data
```

That command should work with Windows, Mac or Linux and will install the stable version.

Alternativelly, you can install the development version with this command:

```
pip install -U pip git+https://github.com/Cerberus1746/PandaCoreData.git@development
```

Also, this badge attests if the unit tests in the development branch are working, if it's red, something in the package is not working and you are better off using just the stable version. It's automatically updated on each commit. Also, don't worry, I get automatically notified if the tests fail.

Actually I advise you to always use the stable version. Unless you want to test new features and such.

You can check the docs of the development version here: https://pandacoredata.readthedocs.io/en/development/

And the badge that checks if the documentation of the development version is working is this:

1.2 Quick Start

Once installed, the following command is available:

```
panda_core_data_commands -o folder_name -re json
```

Which will create all the necessary folder structure and a example *Model* together with their raws inside the supplied *folder_name*. You can also set the *-re* argument as either *json* or *yaml* which will create an example raw based on the supplied extension.

1.3 Data Types

The library has mostly 4 types of data structures, *Model* and *Template*, and then we have their raw files being *raw templates* and *raw models*.

Once you create a class and inherit a *Model* or *Template*, the class is automatically transformed into a dataclass.

1.3.1 Models

Models are pretty much what you will use inside your game to use the data. You can instance multiple times *Model* classes each containing a different set of data and accessing them based on how you created the model. But by default you set the fields like you do with a *dataclass* and the automatic finders will create a model instance with the data from the raw.

For learning purposes. Let's consider you executed the *quick start command* with the folder named *tutorial*. Then go to the file */tutorial/mods/core/models/example_model.py* rename however you like and write it like this:

```
from panda_core_data.model import Model

class Items(Model, data_name="items"):
    name: str
    description: str
    cost: int
```

Mostly, we are just setting the *data_name* parameter to make the **I** in low caps there's more parameters in *_add_into()*. Also remember, if you inherit *Template* or *Model*, the class will turn into a dataclass, so you can instance the model like this for example:

```
Items("Copper", "Fragile material", 1) # The args are in the field order
Items(name="Copper", description="Fragile material", cost=1) # as kwargs
```

But that's not the point of our library, the point is to have easy way to load data from raw files. So let's go to the folder */tutorial/mods/core/raws/models/* and rename the folder *model_name* to the name of your model which in our current case is *items* if you didn't set the param *data_name* the model name will be *Items* with a capital **I** because the library will set the same name as the class.

Since models can be instanced multiple times, it will read all raw files inside the folder that have the same name as the model (if it's inside the folder */mods/core/raws/models/* in this case) and load a instance with the data of the raw.

1.3.2 Raws

The raws are pretty much plain text files that holds data for our instances. The available formats the package support are *yaml* and *json* and soon we will add support for *xml*

So let's go to the file */tutorial/mods/core/raws/models/items/example_model_raw.yaml* rename it to whatever name you'd like, for the tutorial let's name it *copper.yaml* and set it's contents to:

```
data:
    - name: "Copper"
    - description: "Fragile material"
    - value: 1
```

Also, now as in version *0.0.2* the package supports json, so alternatively you can use the example below. The json code would be able to work with yaml extension tho, but, I would advise against it because the pyaml package would attempt to decode a mix of json and yaml code, and yaml is slower than json.

```
{"data": [
    {"name": "Copper"},
    {"description": "Fragile material"},
    {"value": 1}
]}
```

And the data of our instance will be the same as if you were using yaml syntax.

To load the raw you can do like this:

```
raw_path = "/tutorial/mods/core/raws/models/items/copper.yaml"
copper = Items.instance_from_raw(raw_path)
```

Needless to say you need to fix the path to the file. Because I'm not in your computer and I don't know if you use gentoo with a custom kernel having the root folder named *popcorn* (I don't even know if it's possible to change the root folder, but if I could I would totally name it to popcorn).

Also, in this case, the raw file can be anywhere in the disk, or different atoms in your SSD, because, of course, who would still use a disk (me). It can be inside the folder *popcorn/* if you'd like.

But guess what, we don't need to worry to call every single raw or even to import our model inside our game, because we:

1.4 Using Data Core

`DataCore` is the class we use to access all the types, instances and data. It's use is (hopefully) simple. Let's edit the file */tutorial/main.py* to this:

```
from os.path import join, dirname, abspath
from dataclasses import fields
from panda_core_data import data_core

def main():
    # Let's automatically get the folder named Popco- mods, I mean.
    mods_folder = join(dirname(abspath(__file__)), "mods")

    # Templates are something we will cover in the future, but for now
    # let's set them to False
    data_core(mods_folder, templates_folder=False)

    # If we use a for with a model class, we will get all instances of it.
    for instance in data_core.get_model_type("items"):
        print(f"\nValues for: {instance.name}")
        # Remember that I said our class turned into a dataclass?
        # We can iter along the fields now.
        for field in fields(instance):
```

(continues on next page)

(continued from previous page)

```
            print(f"\t{field.name}: {getattr(instance, field.name)}")

if __name__ == '__main__':
    main()
```

This will output the values of our raw file without calling it, without even importing our model and etc etc etc. So much if you like you can create another file in */tutorial/mods/core/raws/models/items/* and the instance will automatically be created. Also, the package will automatically choose the correct parser based on the extension of the raw file. So you are able to mix *json* and *yaml* files together. But please, don't unless you have a good reason for that.

CHAPTER 2

Using the Package with Unity Game Engine

This is a sort of experiment I did while developing this package, which I focused in making a simple working example in Unity and make it work with a build. I executed all necessary functions to check which python interpreter and where the module is being loaded. And they returned that they were loaded from the game build path, which it's important when you ship your game so you don't need to ask for the user to install python together with your game.

However I did not try to execute it in a machine that doesn't have Python, so if you want to use it in production that test must be made.

So in this example we will learn how to:

- Configure a Unity project
- Use Python using Unity
- Learn how to install a package using pip and use inside Unity
- Code C# using python modules and Python.Net

Note: You can't use yaml if you use Unity because PyYaml raises errors, so you have to use json raws.

2.1 Setting a Unity Project

For this we are using Unity 2019.02.0f1 but it would probably work with any Unity that is able to execute .net framework 4.x.

So, create a new Unity project as normal, for learning purposes let's name the project *core_data*. Then set the API compatibility to framework 4.x you can read how to in this link: https://docs.microsoft.com/en-us/visualstudio/cross-platform/unity-scripting-upgrade?view=vs-2019#enabling-the-net-4x-scripting-runtime-in-unity which will inform a couple more things about the difference between the versions.

Note: I tested with netstandard 2.0 however it did not work because dynamic fields needs a microsoft dll which is not referenced.

2.2 Installing Python.net

The easiest way that I found how to set everything up, is first of all, install python 3.7 64 bits, or 32 depending on the final build you desire and then install virtualenv with this command:

```
pip install virtualenv
```

If you fell with a parachute here. The command above installs packages from the Python Library Index (PyPI) and virtualenv let you create a python interpreter that is not linked to the one installed into the system. So, pretty much we will create a virtualenv, *cd* to the root of your project and run this code:

```
virtualenv Packages/python_net
```

Wait for the command to execute. Then activate the virtualenv with:

```
./Packages/python_net/Scripts/activate
```

Then simply install Python.Net with:

```
pip install pythonnet
```

Lastly create a file inside *core_data/Packages/python_net* named *package.json* with this contents:

```
{
  "name": "com.company.package_name",
  "version": "0.0.1",
  "displayName": "Example",
  "description": "Any Description"
}
```

Unity just needs those fields, the contents can be almost anything, but version and name fields needs to follow the pattern above.

2.3 Installing PandaCoreData

The Python interpreter needs to load the module, but Unity compacts everything, but it doesn't compact anything inside the folder *Assets/StreamingAssets* so create that folder if it doesn't exist, then *cd* to that folder. Then execute this command:

```
pip install --ignore-installed git+https://github.com/Cerberus1746/PandaCoreData.git --install-option="--prefix=absolute_folder"
```

And replace *absolute_folder* with the absolute folder to your streaming assets folder. Wait the command to execute and you are done.

Follow this tutorial *Getting Started* and create the file structure with the command in there but inside *Assets/StreamingAssets* folder. Probably you would need to use the quickstart command like this:

```
./Script/panda_core_data_commands -o . -re json
```

Considering that you are executing that command inside the streaming folder, it will create a *mods* folder using json raws. Again, yaml won't work. Then follow *Getting Started* as normal, of course, you won't need to install again the package. But delete the *main.py* file, you won't be needing it because we use a:

2.4 C# Main File

And here's the final working example, the results will be the same as the *Getting Started* but this file will need to be inside the *Assets* folder, outside *StreamingAssets* folder. For convenience sake, let's call it *main.cs*

```
using UnityEngine;
using Python.Runtime;
using System.IO;
using System.Collections.Generic;

namespace PythonTest {
    public class PythonTest : MonoBehaviour {
        void Start() {
            using(Py.GIL()) {
                // Let's import sys
                dynamic py_sys = Py.Import("sys");

                // We need this so we add the python modules from the
                // streaming assets. Otherwise the module won't load.
                string site_pkg = "Lib\\site-packages";
                py_sys.path.insert(0, Path.Combine(Application.streamingAssetsPath,␣
↪site_pkg));

                // Now we can import all necesary modules for the example.
                dynamic py_panda_core_data = Py.Import("panda_core_data");
                dynamic py_dataclasses = Py.Import("dataclasses");
                dynamic py_builtin = Py.Import("builtins");

                // Now we get the mods folder from streaming assets
                string mods_folder = Path.Combine(Application.streamingAssetsPath,
↪"mods");

                // And now we can use the data_core just like we do in python.
                // List type is automatically converted to python equivalent with␣
↪Pyton.Net
                py_panda_core_data.data_core(mods_folder,
                                             templates_folder: false,
                                             excluded_extensions: new List<string>(){
↪"meta"});

                // Now we iterate along all model instances
                dynamic item_model = py_panda_core_data.data_core.get_model_type(
↪"items");
                foreach(dynamic instance in item_model) {
                    // And we can iterate along all fields
                    foreach(dynamic field in py_dataclasses.fields(instance)) {
                        // And show the field name and field value
                        Debug.Log($"{field.name}: {py_builtin.getattr(instance, field.
↪name)}");
```

(continues on next page)

(continued from previous page)

```
                                }
                            }
                        }
                    }
                }
            }
```

Then attatch this script to the camera or any object that you prefer. Then all you need to do is hit play or build the project if you want.

CHAPTER 3

CoreData

3.1 DataCore

class panda_core_data.**DataCore**(**args*, *name=None*, *replace=False*, ***kwargs*)

Bases: *panda_core_data.data_core_bases.data_model.DataModel*, *panda_core_data.data_core_bases.data_template.DataTemplate*

Class where everything is kept.

__call__(*mods_path*, *core_mod_folder='core'*, *raws_folder='raws'*, *models_folder='models'*, *templates_folder='templates'*, ***kwargs*)

Automatically import all data types based on the paths in the params

Parameters

- **mods_path** (Union[Path, str]) – Absolute root folder to the root mods folder
- **core_mod_folder** (Union[Path, str]) – Name of the core mod folder. The base mod
- **raws_folder** (Union[Path, str]) – Name of the raw folder
- **models_folder** (Union[Path, str]) – Name of the models folder
- **templates_folder** (Union[Path, str]) – Name of the templates folder
- **raw_models_folder** (Path or str or bool) – Name of the raws that are related to the models. Default is 'models_folder' param
- **raw_templates_folder** (Path or str or bool) – Name of the raws that are related to the templates. Default is the 'templates_folder' param

Raises *PCDInvalidPath* – If any of the folders are invalid

3.2 DataTemplate

class panda_core_data.data_core_bases.data_template.**DataTemplate**(**args*, ***kwargs*)

Bases: *panda_core_data.data_core_bases.base_data.BaseData*

all_template_instances
Gets all the template instances

Yield Template returns a generator of all instanced templates

Return type Iterator[*Template*]

all_templates
Get all Template types

Return type Tuple[*Template*]

Returns return a list of template types

get_template_type(*template_name*, ***kwargs*)
Get Data type from a list of all Template types

Parameters

- **template_name** (str) – The name of the Template
- **default** – Default value to be returned if the template type couldn't be found

Return type *Template*

Returns the Template

instance_template(*data_type_name*, *path*, ***kwargs*)
Create a new instance of a Template

Parameters

- **template_type_name** – name of the Template
- **path** (Union[Path, str]) – path to the raw file

Return type *Template*

Returns The instanced Template

recursively_instance_template(*path*, **args*, ***kwargs*)
Instance Template recursively based on the raws inside the folders

Return type List[*Template*]

3.3 DataModel

class panda_core_data.data_core_bases.data_model.**DataModel**(**args*, ***kwargs*)

Bases: *panda_core_data.data_core_bases.base_data.BaseData*

all_model_instances
Gets all the model instances.

Yield Model returns a generator of all instanced templates

Return type Iterator[*Model*]

all_models
Get all Model types

Return type Tuple[*Model*]

Returns return a list of model types

get_model_type(*model_name*, ***kwargs*)
Get Data type from a list of all Model types

Parameters

- **model_name** (str) – The name of the Model
- **default** – Default value to be returned if the model type couldn't be found

Return type *Model*

Returns the Model

instance_model(*data_type_name*, *path*, ***kwargs*)
Create a new instance of a Model

Parameters

- **model_type_name** – name of the Model
- **path** (Union[Path, str]) – path to the raw file

Return type *Model*

Returns The instanced Model

recursively_instance_model(*path*, **args*, ***kwargs*)
Instance Model recursively based on the raws inside the folders

Return type Iterator[*Model*]

3.4 BaseData

class panda_core_data.data_core_bases.base_data.**BaseData**(*excluded_extensions=False*)

classmethod __init_subclass__()
This function checks if a method is lacking inside any class that inherits this, and also automatically creates docstrings into those methods based on the original method

static add_module(*path*)
Automatically import the module from the python file and add it's directory to *sys.path* if it wasn't in there before.

Parameters **path** (Union[Path, str]) – The path to the python file

Return module Returns the imported module

Return type module

static all_datas()
Get all DataType types

Return type List[*DataType*]

Returns return a list of data types

static get_data_type(*data_name*, *data_dict*, *default=None*)
Get Data type from a list of all `DataType` types

Parameters

- **data_name** (`str`) – The name of the DataType
- **default** (`Optional`[`bool`]) – Default value to be returned if the data type couldn't be found

Return type *DataType*

Returns the `DataType`

static instance_data(*data_name*, *get_data_type*, *path*, ***kwargs*)
Create a new instance of a `DataType`

Parameters

- **data_type_name** – name of the DataType
- **path** (`Union`[`Path`, `str`]) – path to the raw file

Return type *DataType*

Returns The instanced `DataType`

recursively_add_module(*path*)
Recursively add a module with `DataType` from the supplied path

Parameters **path** (*str*) – Path to the data module

Return type `List`[`module`]

static recursively_instance_data()
Instance `DataType` recursively based on the raws inside the folders

Return type `List`[*DataType*]

CHAPTER 4

Exceptions

Module containing all custom exceptions used by the package. All exceptions related to this package has the PCD prefix.

Please, if you used one of our methods and it raised an exception that doesn't have our PCD prefix **and** it came from inside our files send us a ticket in this link: https://github.com/Cerberus1746/PandaCoreData/issues

author Leandro (Cerberus1746) Benedet Garcia

exception panda_core_data.custom_exceptions.**PCDDataCoreIsNotUnique**
Exception raised if the data core is not unique

exception panda_core_data.custom_exceptions.**PCDDuplicatedModuleName**
Exception raised if a module with the same name was already imported

exception panda_core_data.custom_exceptions.**PCDDuplicatedTypeName**
Exception raised if a `Model` or `Template` Type already exists with the same name

exception panda_core_data.custom_exceptions.**PCDException**
Parent exception for all other exceptions

exception panda_core_data.custom_exceptions.**PCDFolderIsEmpty**
Exception raised if the folder is empty

exception panda_core_data.custom_exceptions.**PCDInvalidBaseData**
Exception raised if a base doesn't have a method

exception panda_core_data.custom_exceptions.**PCDInvalidPath**
Exception raised if the file is invalid

exception panda_core_data.custom_exceptions.**PCDInvalidPathType**
Exception raised if a invalid path type was requested

exception panda_core_data.custom_exceptions.**PCDInvalidRaw**
Exception raised if a raw is invalid

exception panda_core_data.custom_exceptions.**PCDKeyError**
Same as `KeyError`, but with *PCDException*

exception panda_core_data.custom_exceptions.**PCDNeedsToBeInherited**
Exception raised if a `Model` or `Template` would be called without being inherited fist

exception panda_core_data.custom_exceptions.**PCDRawFileNotSupported**
Exception raised if the package can't read the extension

exception panda_core_data.custom_exceptions.**PCDTypeError**
Exception raised if a invalid type was found

CHAPTER 5

Data Types

5.1 DataType

class panda_core_data.data_type.**DataType**(**args*, ***kwargs*)

Base for all the model types, be it template or model

Internally it uses the TinyDB database

__exit__(*_)

Save fields instance into the raw

__getattr__(*attr_name*)

This is here just to make this method back to the default that was overwritten by TinyDB

Parameters **name** – name of the attribute to get

Return type Any

__iter__()

Iter over all documents from default table.

__len__()

Get the total number of documents in the default table.

```
>>> db = TinyDB('db.json')
>>> len(db)
0
```

Return type int

static **__new__**(*cls*, **_*, *db_file=None*, ***__*)

Method that handles the instancing of the models and templates, this is necessary because dataclasses create a custom __init__ method. Which we doesn't use at all if a raw file is supplied.

__repr__()

Return repr(self).

Return type str

__setattr__
Implement setattr(self, name, value).

static _add_into(*data_type*, *data_type_dict*, ***kwargs*)
You can use the prefix *dataclass_* with a dataclass parameter to configure it. They can be found in this link: https://docs.python.org/3/library/dataclasses.html#dataclasses.dataclass

For example:

```
from panda_core_data.model import Model

class ModelName(Model, dataclass_init=False):
    out: int

    def __init__(self, num):
        self.out = num ** num
```

Will make the library not create a *__init__* method and use yours instead.

Parameters

- **data_type** (*DataType*) – class type to be added
- **template_name** (*None or str*) – The name of the template, if not supplied, the class name is used.
- **dependency_list** (*list[str]*) – Template to be used as dependency.

add_dependencies()
Add all dependencies for the model

has_dependencies
If the model has any dependencies

Return type bool

Returns If the instance have dependencies or not.

keys()
Iter over all documents from default table.

load_db(*db_file*, **init_args*, *default_table='data'*, ***kwargs*)
Method that load raw files and assign each field to an attribute.

Parameters

- **db_file** (Union[Path, str]) – Path to a raw file
- **storage** (*tinydb.storages.Storage*) – storage class to be used.
- **default_table** (str) – default main field in the raw file.

save_to_file(*_)
Save fields instance into the raw

5.2 Model

class panda_core_data.model.**Model**(**args*, ***kwargs*)
Bases: *panda_core_data.data_type.DataType*

classmethod **__init_subclass__**(*core_name=None*, ***kwargs*)
Method that automatically registers class types into data_core. You can use the same parameters as *_add_into()*

5.3 Template

class panda_core_data.model.**Template**(**args*, ***kwargs*)
Bases: *panda_core_data.data_type.DataType*

Class that will be used to make ModelTemplates

classmethod **__init_subclass__**(*core_name=None*, ***kwargs*)
Method that automatically registers class types into data_core. You can use the same parameters as *_add_into()*

CHAPTER 6

Storages

Storages are classes based on TinyDB that handles file management, raw parsing and other things like that, if you'd like to create support for more file types, all you have to do is to inherit the *BaseDB* and follow the instructions in the API of that class.

6.1 Directory Utils

panda_core_data.storages.**auto_convert_to_pathlib**(*path*)
Check if the path is valid and automatically convert it into a Path object

Parameters **path** (Union[Path, str]) – source folder

Return type Path

Returns The Path object

Raises ***PCDInvalidPath*** – If the file is invalid

panda_core_data.storages.**get_extension**(*path*)
Get file extension from the path

Parameters **path** (Union[Path, str]) – path to a file or extension

Return type str

Returns The file extension

panda_core_data.storages.**get_raw_extensions**()
Get all available extensions the package supports

Return type List[str]

Returns A list of available extensions

panda_core_data.storages.**get_storage_from_extension**(*extension*)
Returns the storage based on the file extension

Return type tinydb.storages.Storage

Returns Returns a storage object that handles the raw file

panda_core_data.storages.**is_excluded_extension**(*path*, *exclude_ext*)
Check if the file has an ignored extension

Parameters

- **path** (Union[Path, str]) – source folder
- **exclude_ext** (List[str]) – If the path should be from a folder or file

Return type bool

Returns returns True if it's a excluded extension, False otherwise

panda_core_data.storages.**raw_glob_iterator**(*path*, *excluded_ext=False*)
Iterate along the path yielding the raw file.

Yields The file

Return type Iterator[Path]

6.2 YamlDB

class panda_core_data.storages.yaml_db.**YamlDB**(**args*, ***kwargs*)
Parser storage class used to read yaml files

__init__(**args*, ***kwargs*)
Open file as Data Base

Parameters **str** (*path*) – path pointing to a yaml file

read()
Method used by TinyDB to read the file

Return type List[Dict[str, Any]]

write(*data*)
Write the current state of the database to the storage.

Any kind of serialization should go here.

Parameters **data** (*dict*) – The current state of the database.

6.3 JsonDB

class panda_core_data.storages.json_db.**JsonDB**(*path*, ***kwargs*)

read()
Read the last stored state.

Any kind of deserialization should go here. Return None here to indicate that the storage is empty.

Return type dict

write(*data*)
Write the current state of the database to the storage.

Any kind of serialization should go here.

Parameters **data** (*dict*) – The current state of the database.

6.4 BaseDB

class panda_core_data.storages.base_db.**BaseDB**(*path*, ***kwargs*)

Base storage class that reads which extensions are available to feed the path handling functions

To create a new storage, you will need to inherit this class, create a *extensions* variable containing a list of extensions the storage will support for example:

```
extensions = ["yml", "yaml"]
```

then implement a *read* and *write* method using the methods *base_read()* and *base_write()* all you need to do is follow the instructions contained in them

__init__(*path*, ***kwargs*)

Create a new instance

Parameters **path** (*str*) – Path to file

classmethod **__init_subclass__**()

Automatically generate an extension list containing the available raw extensions available together with their storage

base_read(*load_method*, *use_handle*)

Base method used by children classes to read the file and transforms the string into a list of dictionaries, a good example of this method is the built in python `json.load()` however, since it needs a string as an input (or handler) you would need to set the parameter use_handler so the string, which is the contents of the raw file, will be passed to that method. For example the read method of our yaml parser:

```
def read(self):
    return self.base_read(yaml.safe_load, True)
```

And since the function `yaml.safe_load()` needs a string as an input, we set use_handle to True.

An example of list of dictionaries would be like this:

```
{"data": [
    {
        'field_name': 'value',
        'another_field': 10,
    },
    {
        'field_name': 'value',
        'another_field': 10,
    },
]}
```

The dict keys are fields of a `dataclass` and the value, well, values

Parameters

- **load_method** (`Callable`) – method used to transform the raw file into a list of dictionaries
- **use_handle** (`bool`) – TinyDB offers a handle (More specifically, the handle of the class `JSONStorage`) to load the file and turn into a string automatically if you'd like to use it, just set this parameter to True

Return type `List[Dict[str, Any]]`

Returns The generated data

base_write(*write_method*, *data*, *use_handle*)
Transforms the data dictionary to a raw representation

Parameters

- **write_method** (Callable) – method used to transform the raw file into a list of dictionaries
- **data** (List[Dict[str, Any]]) – data dictionary

CHAPTER 7

Utils

created 2019-07-29

author Leandro (Cerberus1746) Benedet Garcia

panda_core_data.utils.**check_if_valid_instance**(*an_object*, *the_type*)

Check if *an_object* is a instance from *the_type*.

Parameters

- **an_object** (Any) – source object
- **the_type** (Any) – type object

Raises ***PCDTypeError*** – if the type is not a instance or wrong type

CHAPTER 8

Change Logs

8.1 0.0.6

8.1.1 Added

- *sphinx_autodoc_typehints* is now a dependency fr the docs

8.1.2 Changed

- All functions and methods now have type annotations and the documentation uses them to determine the type

8.2 0.0.5

8.2.1 Changed

- Everytime a *DataType* is instanced, it will be added to the attatched *DataCore*. The behaviour was that the instance would be added only by calling *instance_model()* and *instance_template()*. Also tests were changed to make sure it happens.
- All exceptions now inherit *PCDException*.
- `__init__()` methods are not overwritten anymore.

8.3 0.0.4

8.3.1 Added

- Support for tox

8.3.2 Changed

- Fixed the installer to behave better, which could derp while reading some files.

8.4 0.0.3

8.4.1 Changed

- Improved compatibility with Python.Net

8.4.2 Fixed

- Fixed a bug with the *_get_core()* method

8.5 0.0.2

8.5.1 Added

- Support for Json was added
- Now, it's possible to save `Model` or `Template` to the raw with the `save_to_file()` method
- `instance_model()` and `instance_template()` and *older_contents* were added.
- *panda_core_data_commands* has a new option *-re* which you can choose between *json* or *yaml*
- `PCDInvalidRaw` and `PCDDuplicatedModuleName` exceptions was created
- Package *Storage* was created

8.5.2 Changed

- `auto_convert_to_pathlib()` do not need the *is_file* parameter anymore
- `panda_core_data.DataCore` has a new parameter *excluded_extensions*
- Both exceptions *PCDFolderNotFound* and *PCDFileNotFound* was merged into `PCDInvalidPath`

8.6 0.0.1

8.6.1 Added

- Package released
- Basic yaml reading which their contents is loaded to a python *dataclass*

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D