

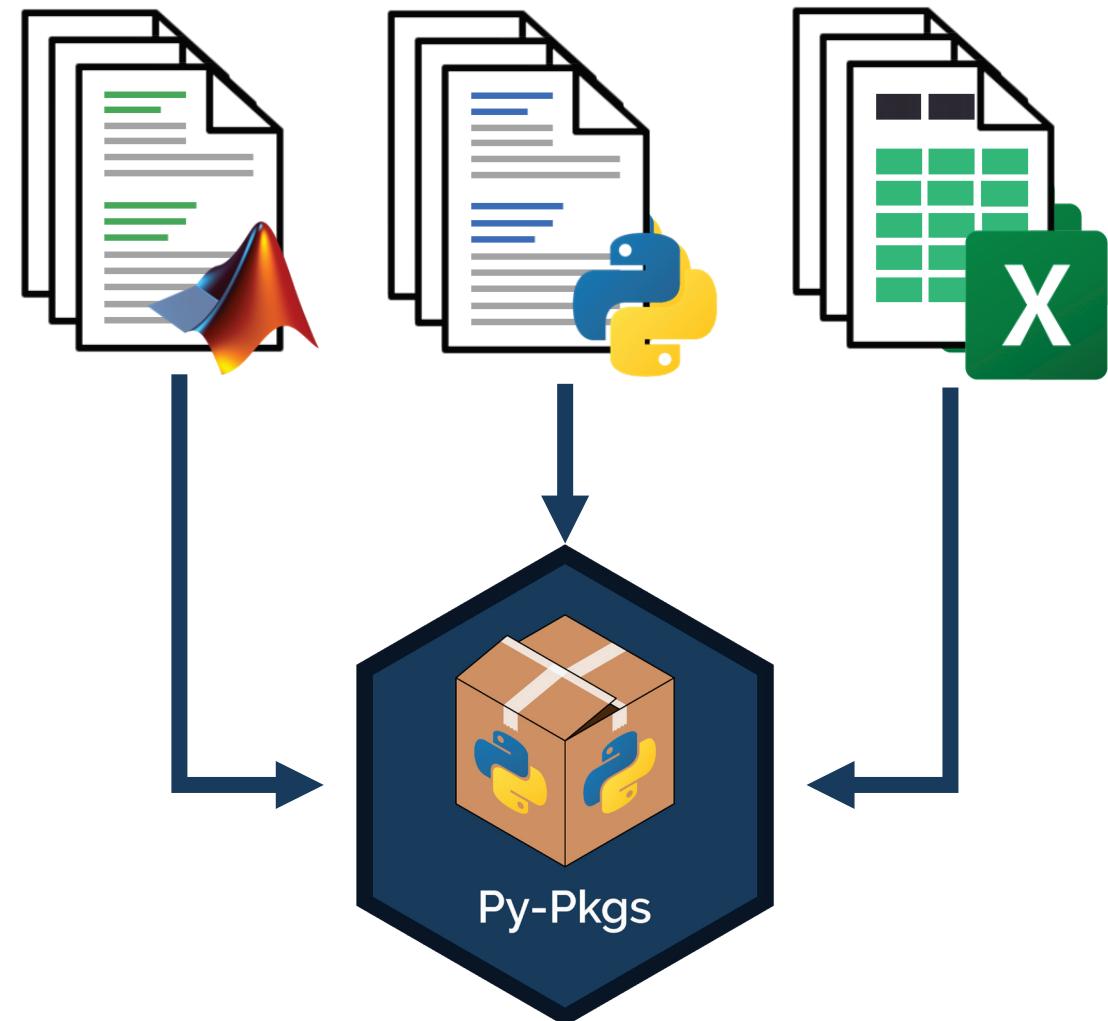
# LibICE-post: User guide

*Post-processing of experimental and numerical data for ICE applications*

Federico Ramognino

# Why libICEpost...?

- No common methodology was present for processing of results (experimental or CFD simulations), leading to:
  - Redundancy and duplication of code
  - Multiple codes and software
  - Risk of errors
  - Loss of efficiency and knowledge
- Need for a **standardized framework** to be shared within the research group
- Organizing all methodologies in a single **python package**
- ...why **python**?
  - ✓ Free (no licence required)
  - ✓ Strong community (many packages available)
  - ✓ Easy to use
  - ✓ Versatile
  - ✓ Not OS-dependent



# What is inside libICEpost

- **Scope of the package:**
  - Implementation methodologies for pre/post-processing of data from internal combustion engines applications
- **Main features:**
  - ✓ Modeling of mixtures, with their thermodynamic properties
  - ✓ Modeling of internal combustion engines for processing heat release rate and heat transfer from pressure traces
  - ✓ Input/output interfaces for processing OpenFOAM results
- **Requirements:** python>=3.11

```
user@localhost: ~  
  
(base) user@localhost:~$ pip install libICEpost
```



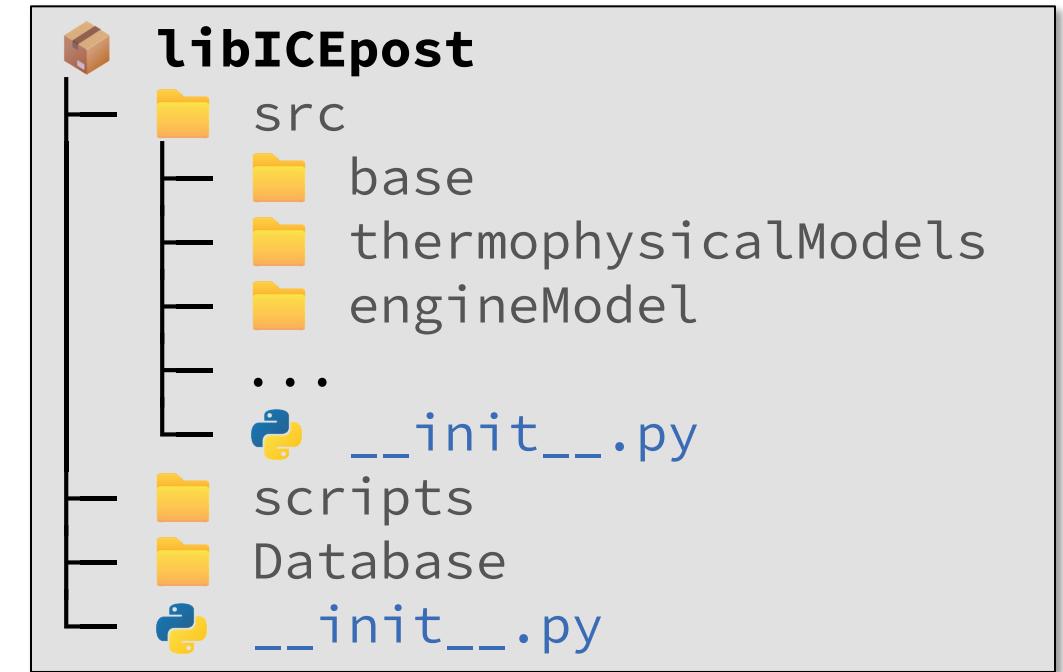
**PyPI:** <https://pypi.org/project/libICEpost/>



**GitHub:** <https://github.com/RamogninoF/LibICE-post>

# Structure of the package

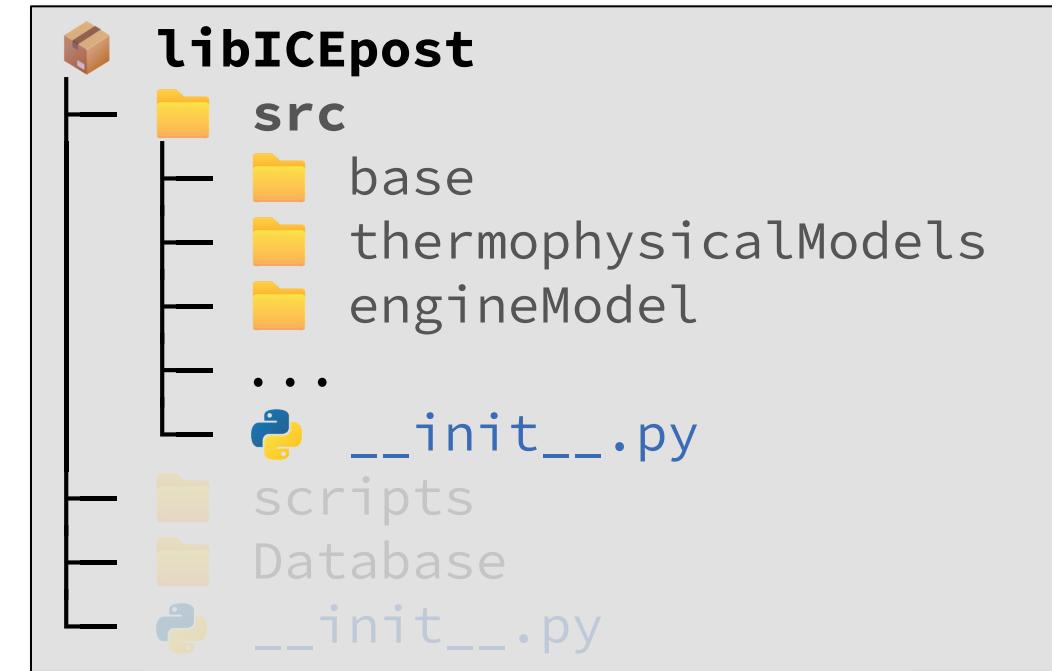
- Three main subpackages:



# Structure of the package

- **Three main subpackages:**

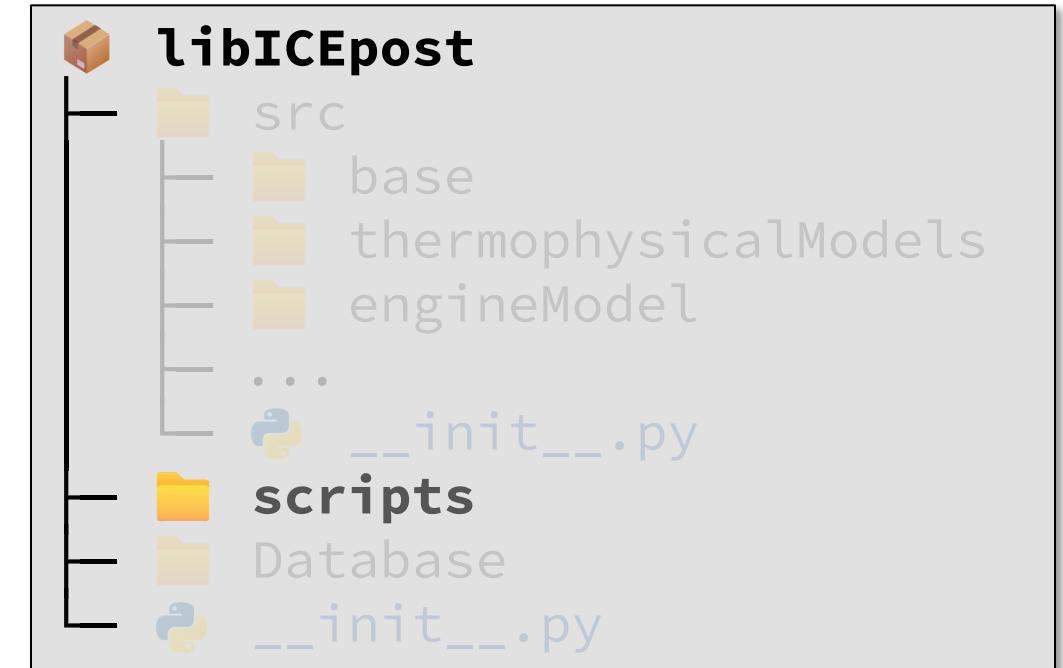
- **src** – Core of the library, contains all **functions and classes**. Major groups are:
  - base – generic-use classes and functions
  - thermophysicalModels – modeling of chemistry and thermophysical properties of mixtures
  - engineModel – models for description of ICE



# Structure of the package

- **Three main subpackages:**

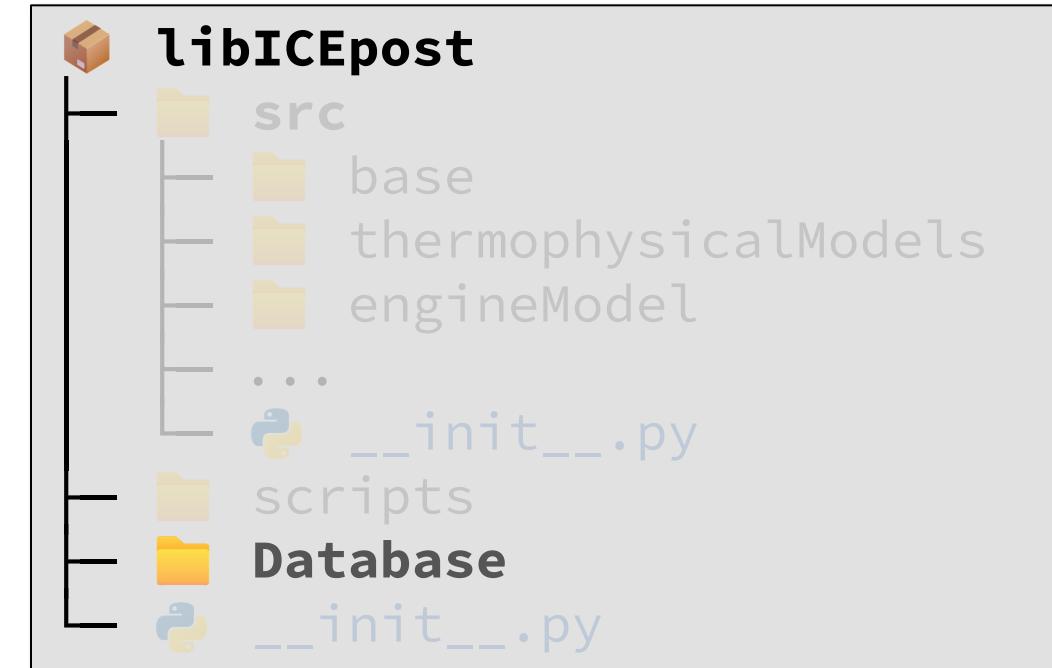
- **src** – Core of the library, contains all functions and classes. Major groups are:
  - base – generic-use classes and functions
  - thermophysicalModels – modeling of chemistry and thermophysical properties of mixtures
  - engineModel – models for description of ICE
- **scripts** – Contains **executable scripts**, where automated workflows can be implemented



# Structure of the package

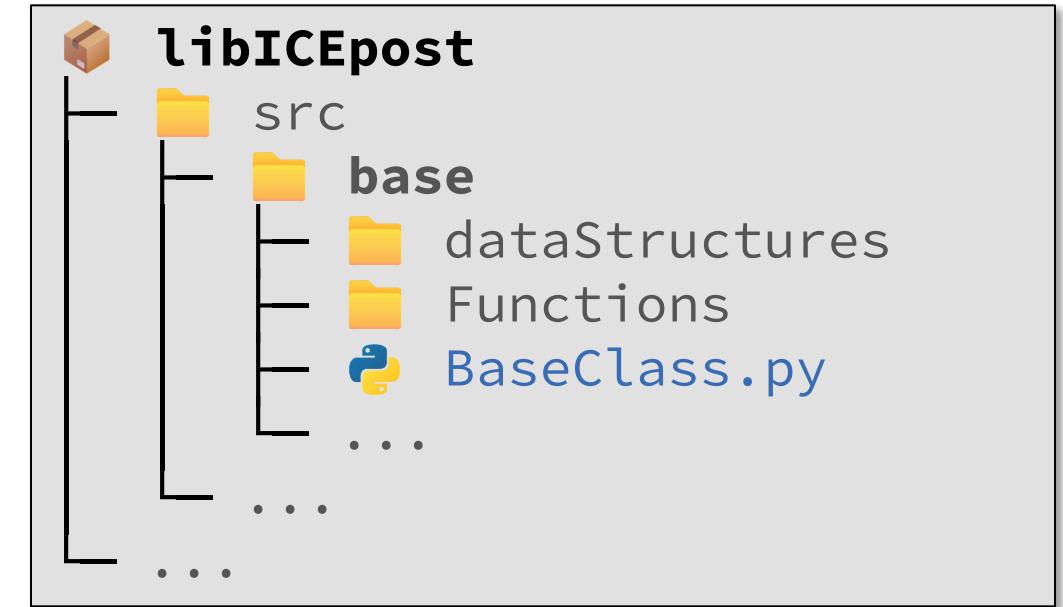
- **Three main subpackages:**

- **src** – Core of the library, contains all functions and classes. Major groups are:
  - base – generic-use classes and functions
  - thermophysicalModels – modeling of chemistry and thermophysical properties of mixtures
  - engineModel – models for description of ICE
- **scripts** – Contains executable scripts, where automated workflows can be implemented
- **Database** – A built-in database with useful data (periodic table, molecules, fuels, thermophysical properties, etc.)



# libICEpost.src.base

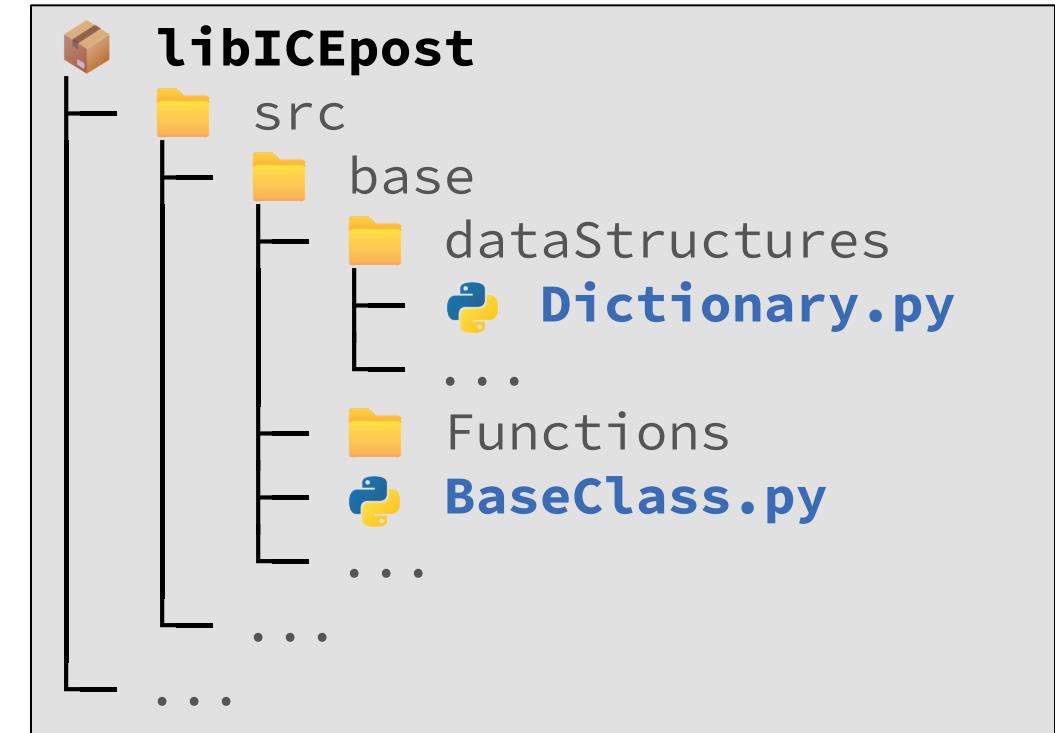
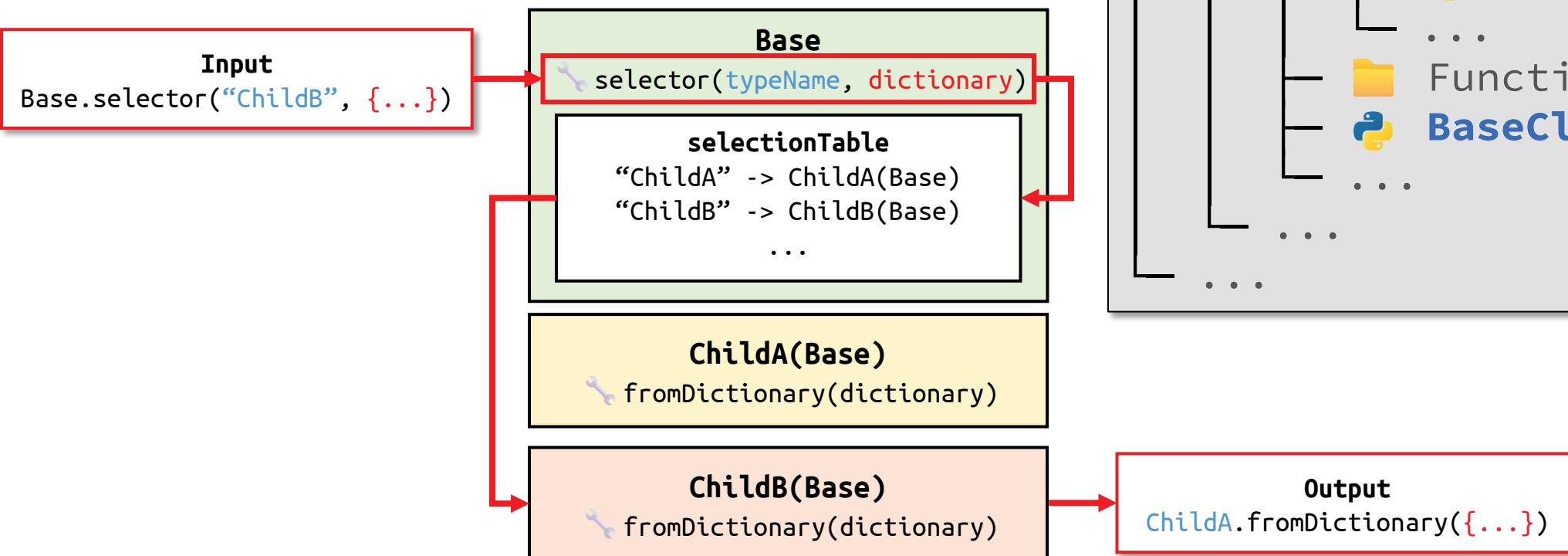
- Generic-use **classes** (data-structures, signal filters, etc.) and **functions** (I/O, OpenFOAM interfaces, etc.)



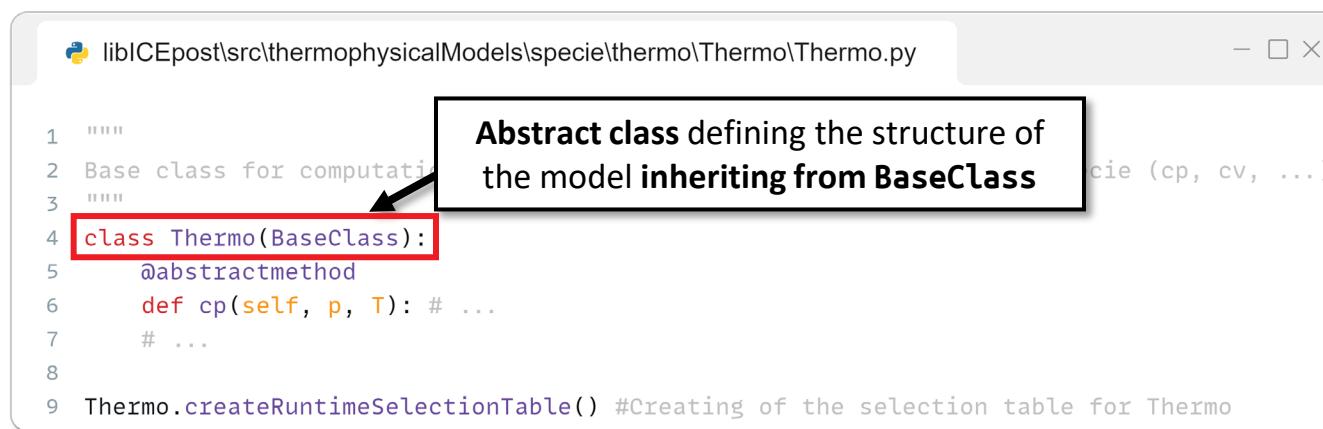
# Input/Output and run-time selection

9

- Generic-use classes (data-structures, signal filters, etc.) and functions (I/O, OpenFOAM interfaces, etc.)
  - Implementation of **I/O** and **run-time selection** of models following OpenFOAM rationale



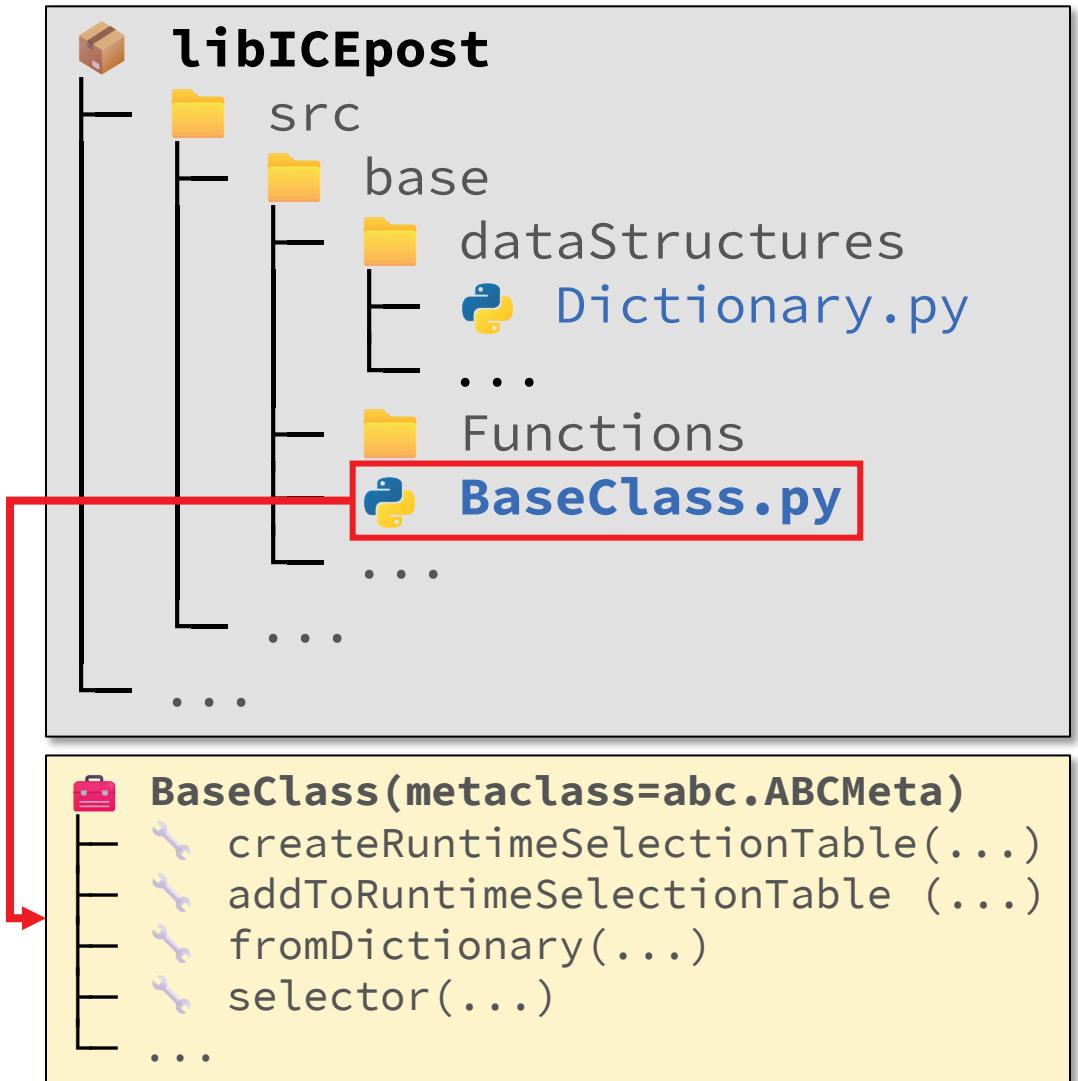
- **BaseClass** – Virtual Base Class with implementation of run-time selection for handling model choice
  - Used to give abstract-base-class functionalities to classes defining the structure of models



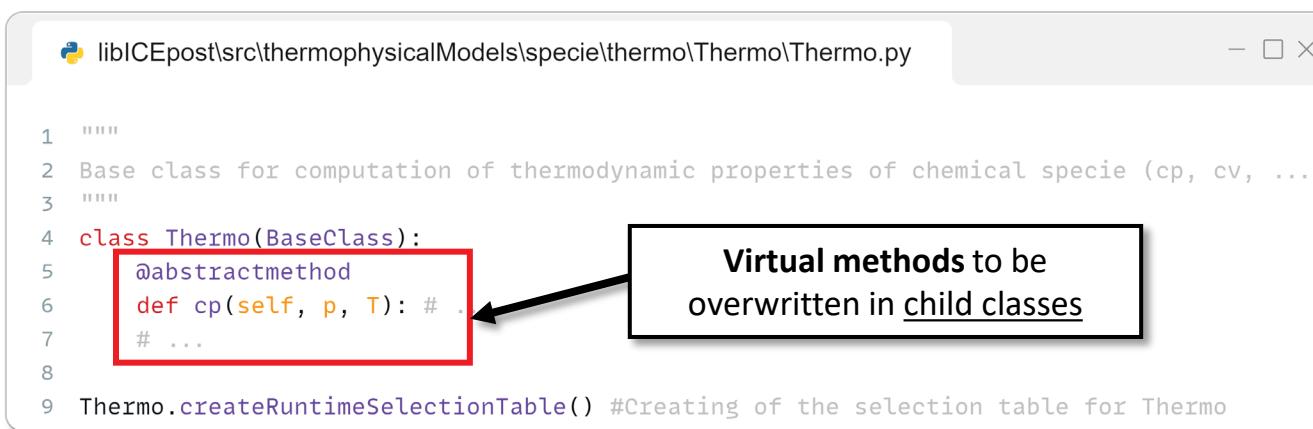
libICEpost\src\thermophysicalModels\specie\thermo\Thermo.py

```
1 """
2 Base class for computation
3 """
4 class Thermo(BaseClass):
5     @abstractmethod
6     def cp(self, p, T): # ...
7     # ...
8
9 Thermo.createRuntimeSelectionTable() #Creating of the selection table for Thermo
```

**Abstract class defining the structure of the model inheriting from BaseClass**



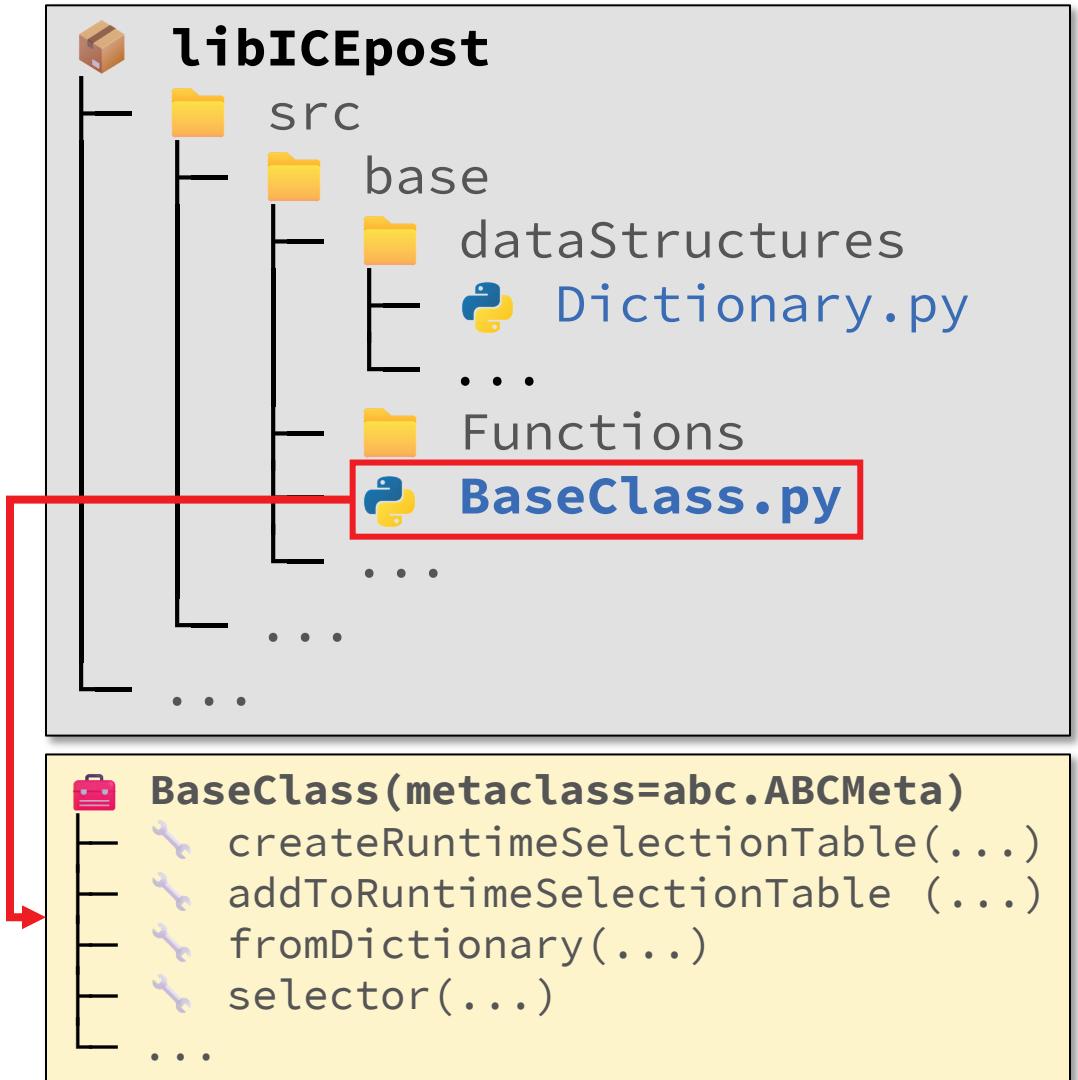
- **BaseClass** – Virtual Base Class with implementation of run-time selection for handling model choice
  - Used to give abstract-base-class functionalities to classes defining the structure of models



```
libICEpost\src\thermophysicalModels\specie\thermo\Thermo.py
```

```
1 """
2 Base class for computation of thermodynamic properties of chemical specie (cp, cv, ...)
3 """
4 class Thermo(BaseClass):
5     @abstractmethod
6     def cp(self, p, T): # .
7     # ...
8
9 Thermo.createRuntimeSelectionTable() #Creating of the selection table for Thermo
```

VIRTUAL METHODS TO BE OVERWRITTEN IN CHILD CLASSES

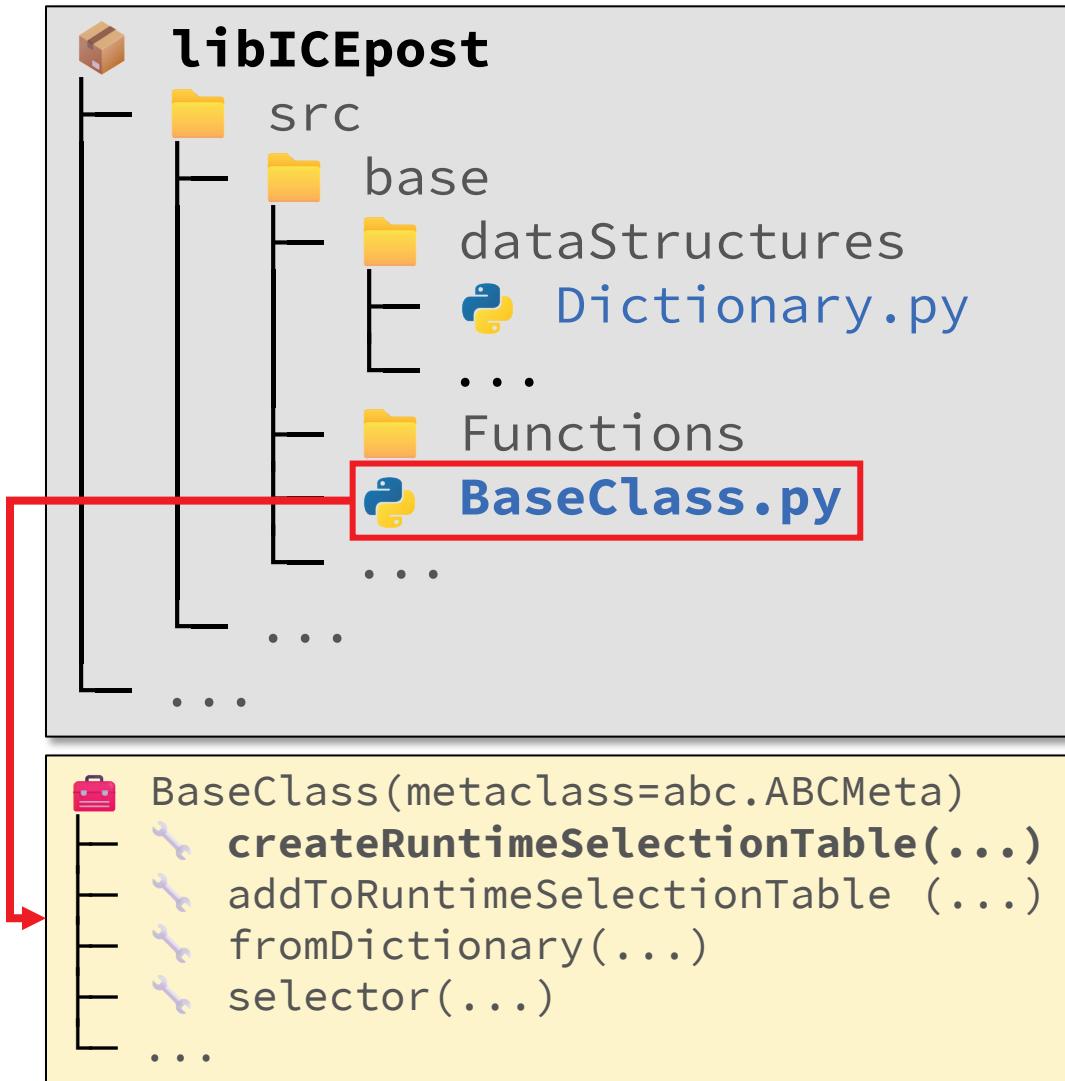


# Run-time selection – BaseClass

- **BaseClass** – Virtual Base Class with implementation of run-time selection for handling model choice
    - Used to give abstract-base-class functionalities to classes defining the structure of models
    - **Creation of selection-table** for the **VBC** defining the common structure of the specific model

```
1 """
2 Base class for computation of thermodynamic properties of chemical specie (cp, cv, ...)
3 """
4 class Thermo(BaseClass):
5     @abstractmethod
6     def cp(self, p, T): # ...
7     # ...
8
9 Thermo.createRuntimeSelectionTable() #Creating of the selection table for Thermo
```

Creation of the **selection table** for class Thermo



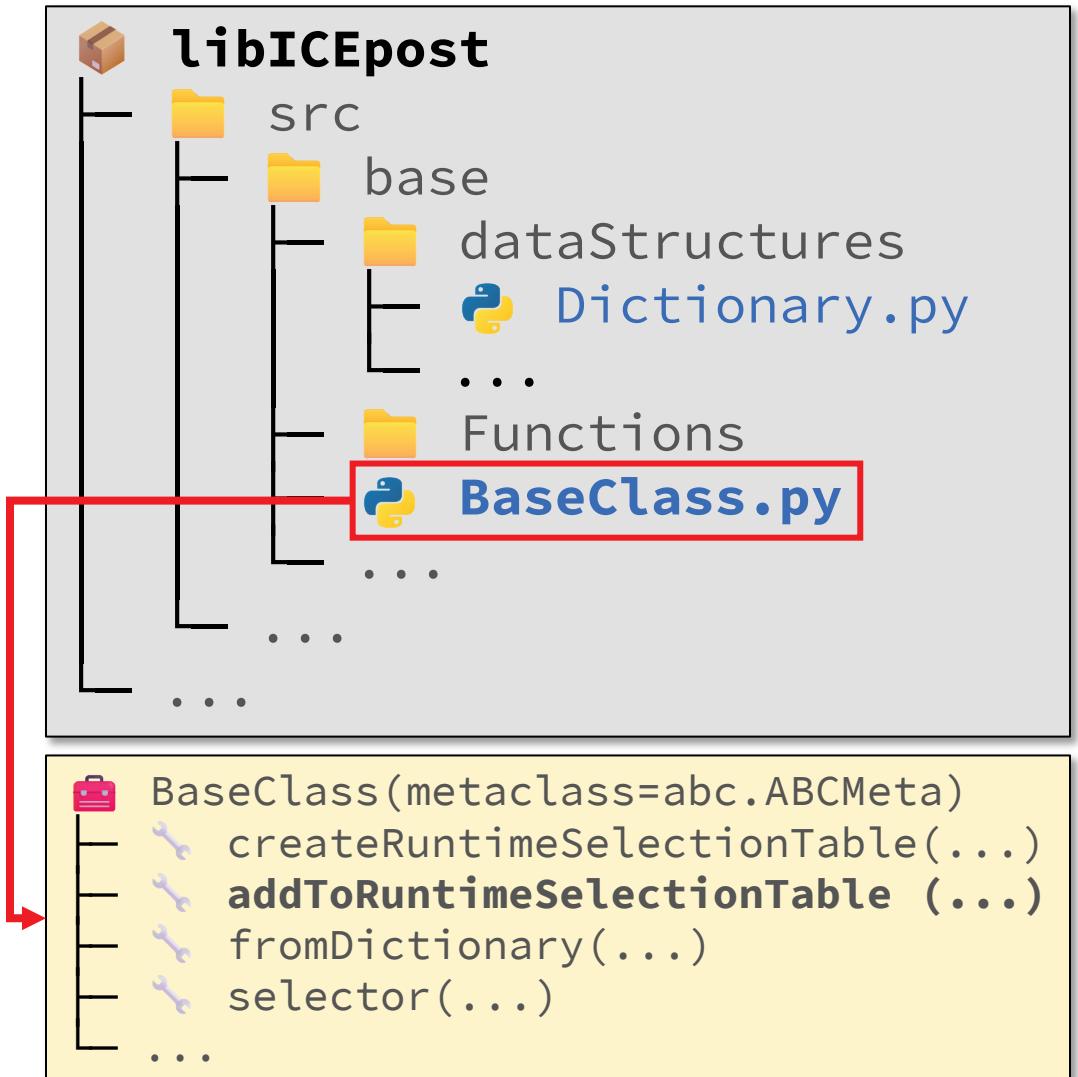
- **BaseClass** – Virtual Base Class with implementation of run-time selection for handling model choice
  - Used to give abstract-base-class functionalities to classes defining the structure of models
  - Creation of selection-table for the VBC defining the common structure of the specific model
  - Deriving concrete child classes from the VBC with the desired model implementation, adding them to its selection table

```
libICEpost\src\thermophysicalModels\specie\thermo\Thermo\janaf7.py
```

```

1  from .Thermo import Thermo      #Importing the (virtual) base class
2
3  # Computation of thermophysical properties based on polynomial fits to
4  class janaf7(Thermo):          # Inheriting from abstract class Thermo
5      @classmethod
6      def fromDictionary(cls,dictionary): ...
7      @classmethod
8      def cp(self, p, T): ...
9
10 Thermo.addToRuntimeSelectionTable(janaf7) #Adding janaf7 to the table of Thermo

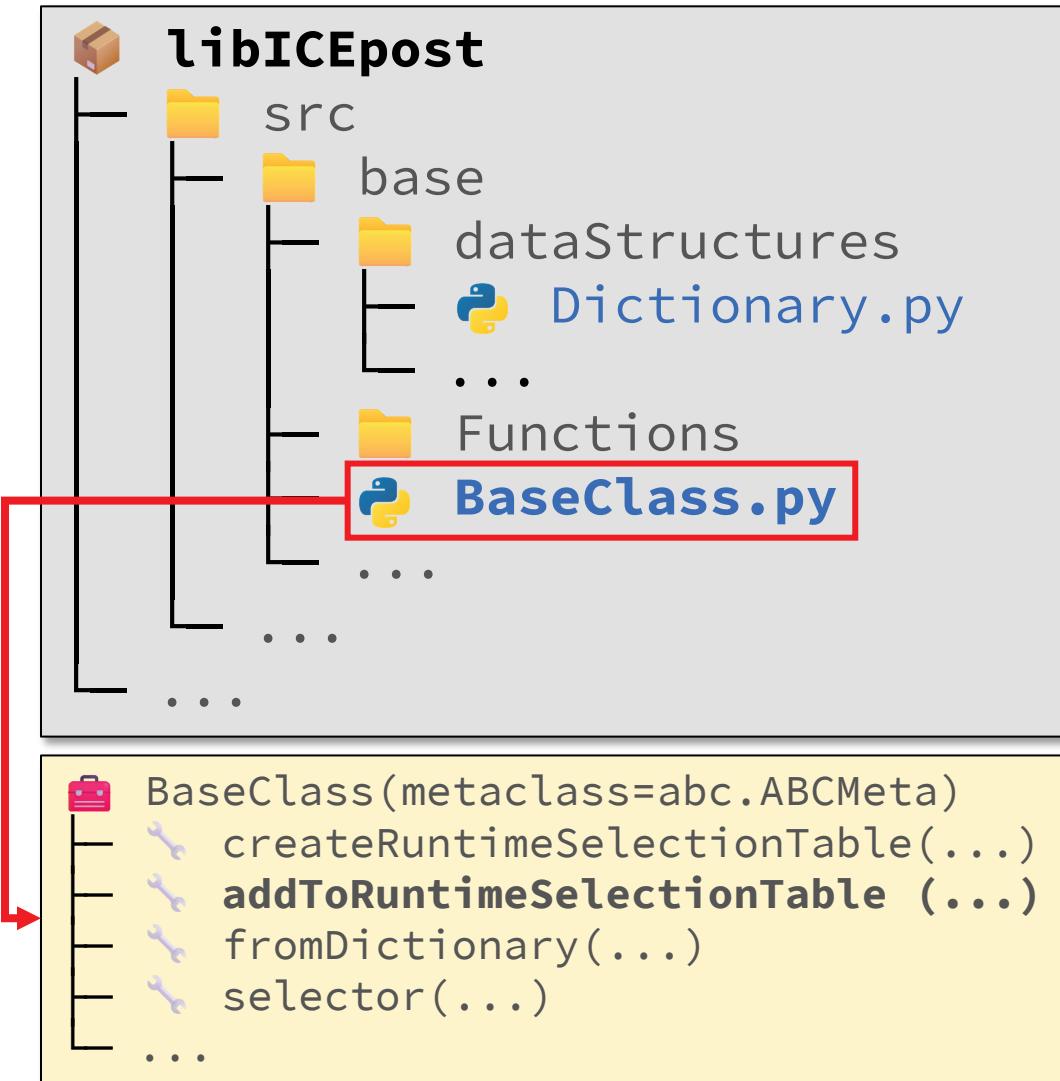
```



- **BaseClass** – Virtual Base Class with implementation of run-time selection for handling model choice
    - Used to give abstract-base-class functionalities to classes defining the structure of models
    - Creation of selection-table for the **VBC** defining the common structure of the specific model
    - Deriving **concrete child classes** from the VBC with the desired **model implementation**, adding them to its selection table

libICEpost\src\thermophysicalModels\specie\thermo\Thermo\janaf7.py

```
1 from .Thermo import Thermo      #Importing the (virtual) base class
2
3 # Computation of thermophysical properties with NASA 7-coefficient polynomials.
4 class janaf7(Thermo):
5     @classmethod
6     def fromDictionary(cls,dictionary):
7         @classmethod
8         def cp(self, p, T): ...
9
10 Thermo.addToRuntimeSelectionTable(janaf7) #Adding janaf7 to the table of Thermo
```

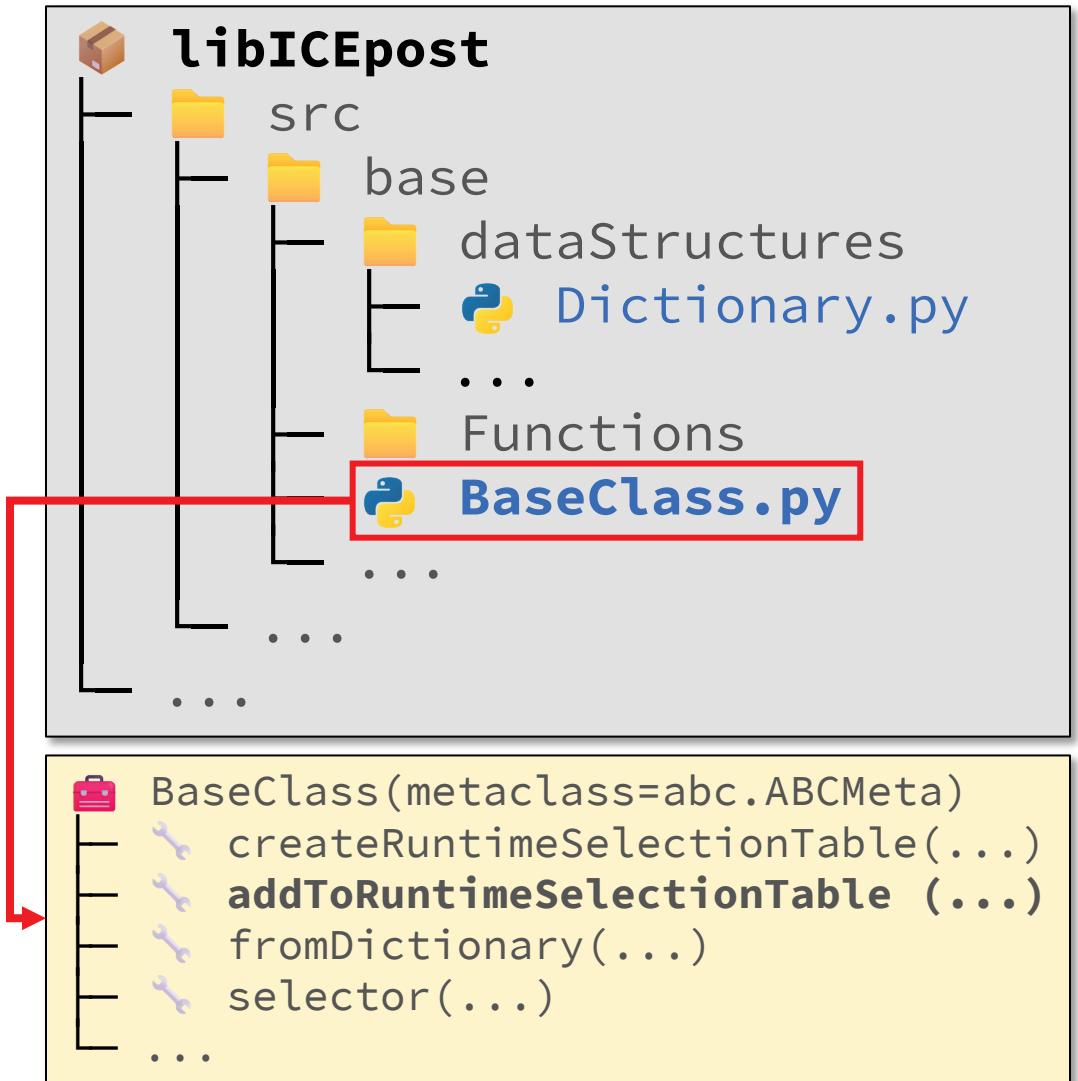


- **BaseClass** – Virtual Base Class with implementation of run-time selection for handling model choice
  - Used to give abstract-base-class functionalities to classes defining the structure of models
  - Creation of selection-table for the VBC defining the common structure of the specific model
  - Deriving concrete child classes from the VBC with the desired model implementation, adding them to its selection table

```
libICEpost\src\thermophysicalModels\specie\thermo\Thermo\janaf7.py
```

```
1 from .Thermo import Thermo      #Importing the (virtual) base class
2
3 # Computation of thermophysical properties with NASA 7-coefficient polynomials.
4 class janaf7(Thermo):
5     @classmethod
6     def fromDictionary(cls,dictionary): ...
7     @classmethod
8     def cp(self, p, T): ...
9
10 Thermo.addToRuntimeSelectionTable(janaf7) #Adding janaf7 to the table of Thermo
```

Adding janaf7 to the selection table of Thermo

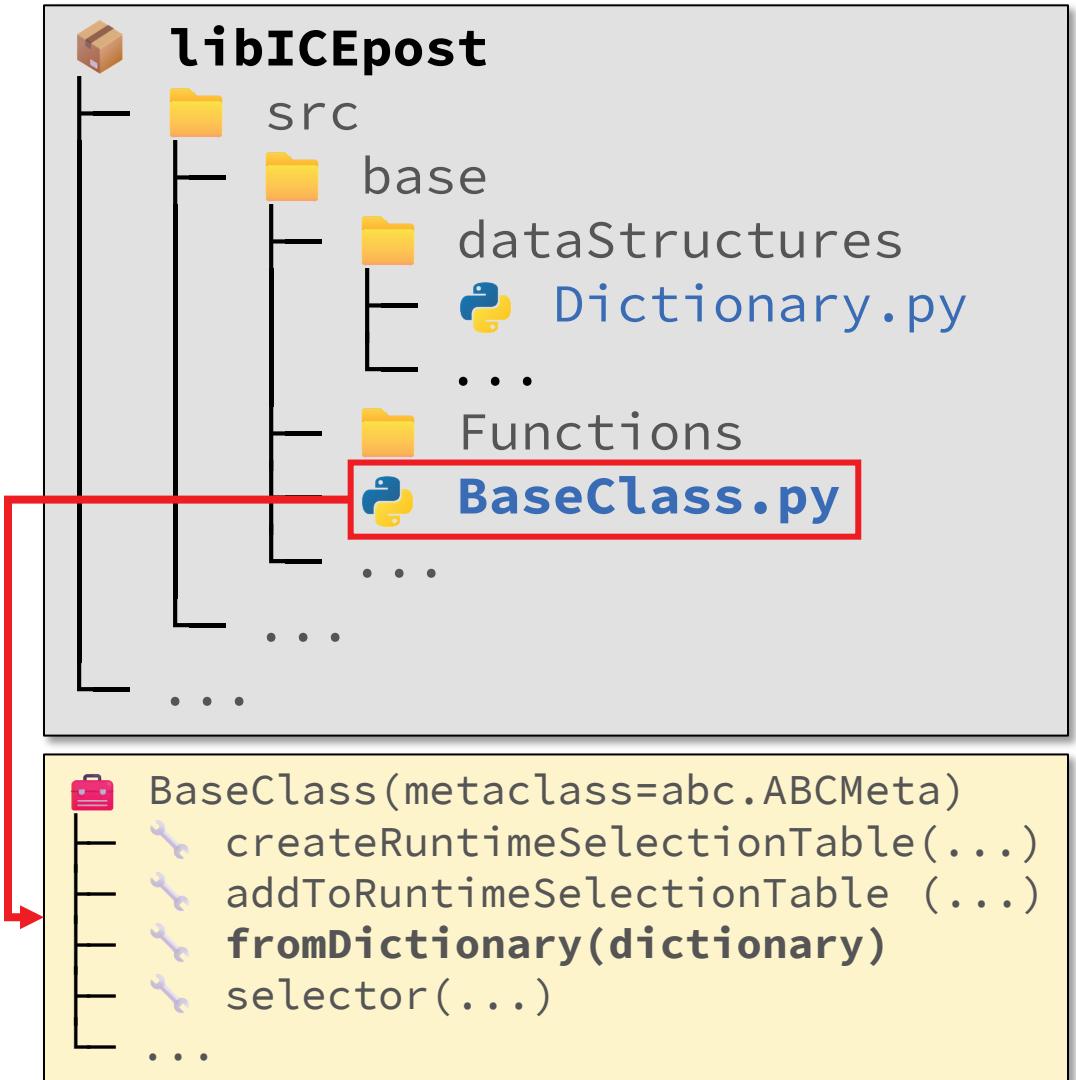


- **BaseClass** – Virtual Base Class with implementation of run-time selection for handling model choice
  - Used to give abstract-base-class functionalities to classes defining the structure of models
  - Creation of selection-table for the VBC defining the common structure of the specific model
  - Deriving concrete child classes from the VBC with the desired model implementation, adding them to its selection table
  - Defining a **fromDictionary** method, which is used during the run-time selection for construction from a dictionary

libICEpost\src\thermophysicalModels\specie\thermo\Thermo\janaf7.py

```
1 from .Thermo import Thermo      #Importing the (virtual) base class
2
3 # Computation of thermophysical properties with NASA 7-coefficient polynomials.
4 class janaf7(Thermo):
5     @classmethod
6     def fromDictionary(cls,dictionary): #...
7     @classmethod
8     def cp(self, p, T): #...
9
10 Thermo.addToRuntimeSelectionTable(janaf7) #Adding janaf7 to the table of Thermo
```

Method for construction  
from a dictionary entry

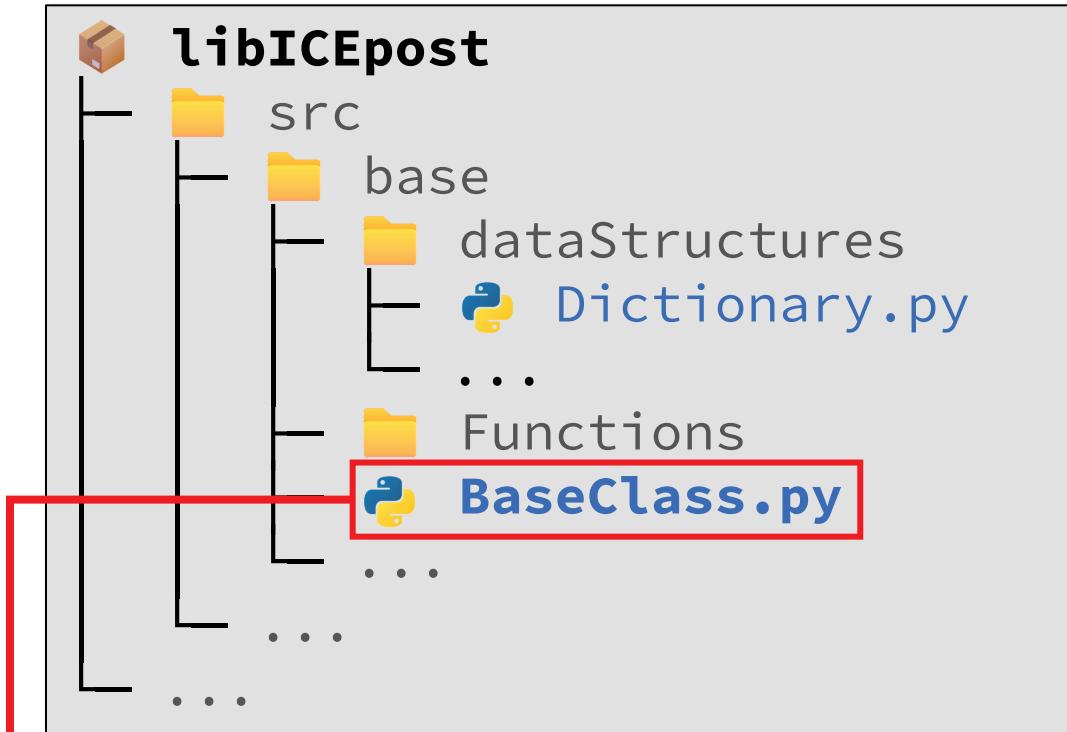


- **BaseClass** – Virtual Base Class with implementation of run-time selection for handling model choice
  - Used to give abstract-base-class functionalities to classes defining the structure of models
  - Creation of selection-table for the VBC defining the common structure of the specific model
  - Deriving concrete child classes from the VBC with the desired model implementation, adding them to its selection table
  - Defining a fromDictionary method, which is used during the run-time selection for construction from a dictionary
  - Selection of the model with a common interface (selector), using the model's name and a dictionary, which is fed to the fromDictionary classmethod of the required child-class

```
test.py
```

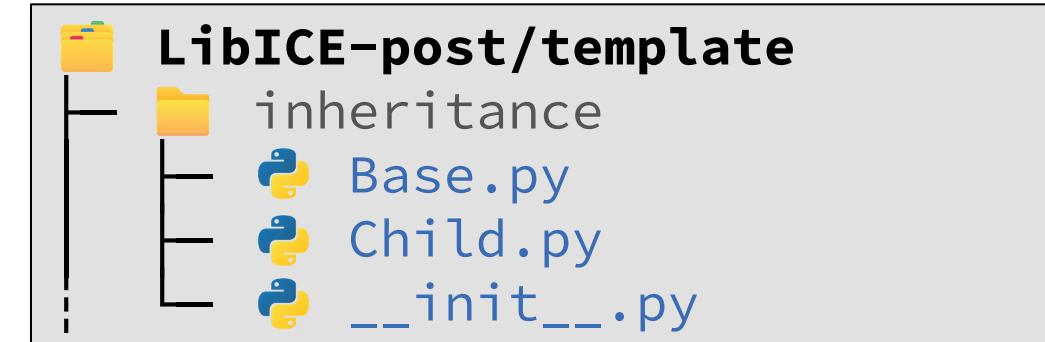
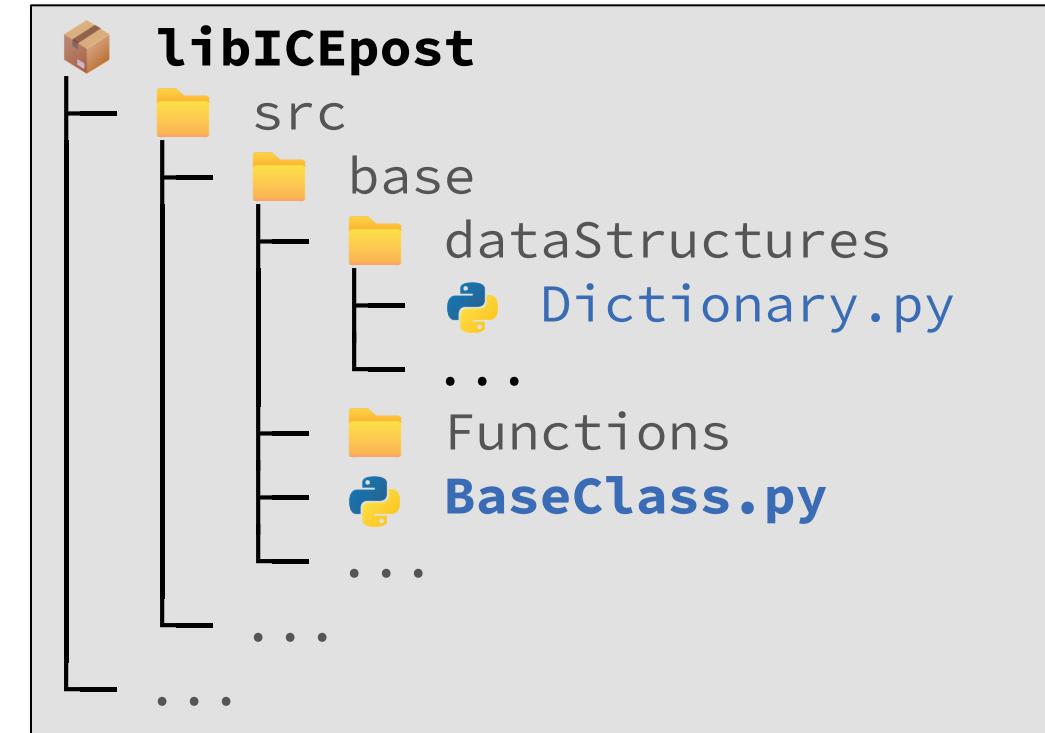
```
1 from libICEpost.thermophysicalModels.specie.thermo.Thermo import Thermo #Loading Thermo
2 dictionary = {
3     "Rgas": 287, # J/kgK
4     "gamma": 1.4, # cp/cv ratio
5 }
6 thermo = Thermo.selector("constantCP", dictionary) #Construction from selector
```

Construction of **constantCP** class from dictionary

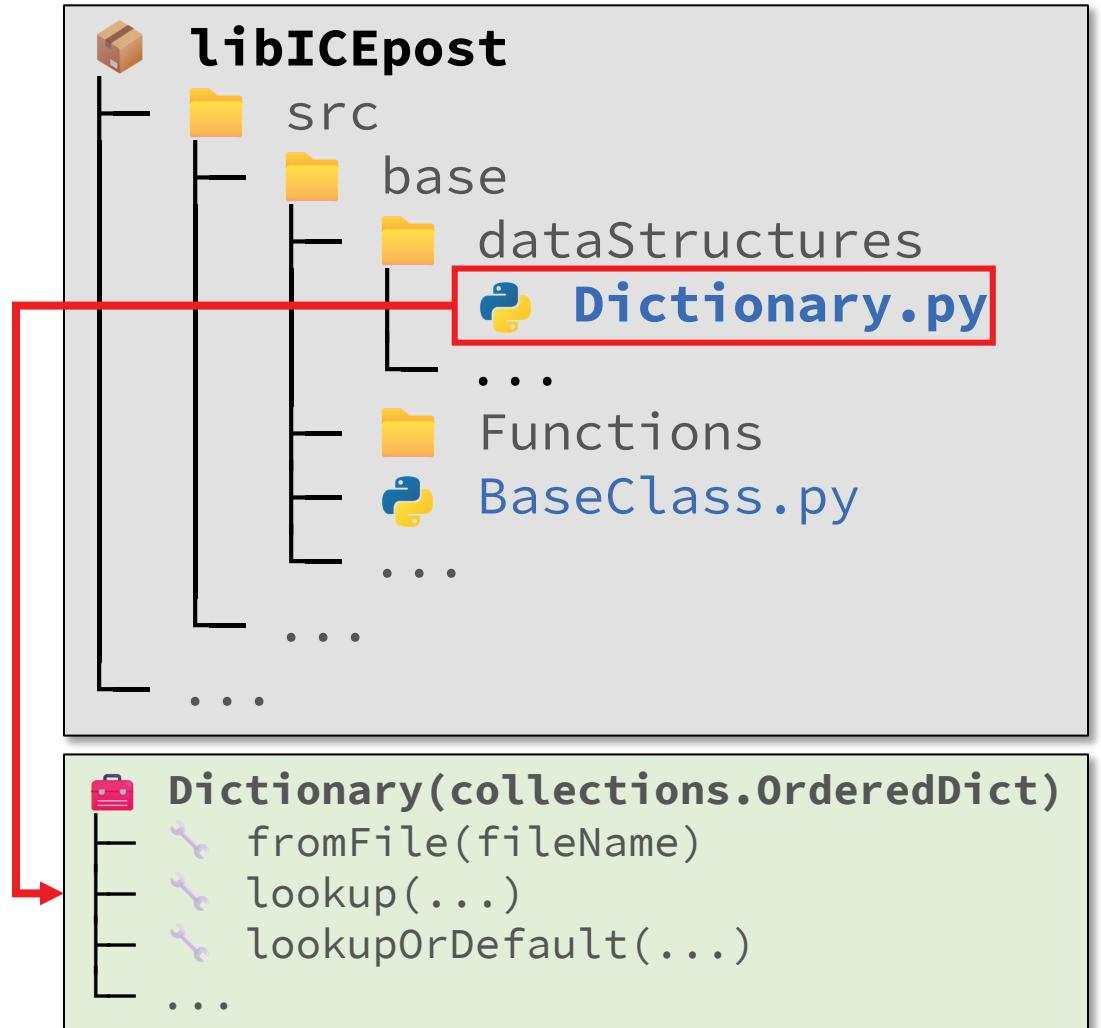


**BaseClass**(metaclass=abc.ABCMeta)  
    createRuntimeSelectionTable(...)  
    addToRuntimeSelectionTable (...)  
    **fromDictionary(dictionary)**  
    **selector(typeName, dictionary)**  
...

- **BaseClass** – Virtual Base Class with implementation of run-time selection for handling model choice
  - Used to give abstract-base-class functionalities to classes defining the structure of models
  - Creation of selection-table for the VBC defining the common structure of the specific model
  - Deriving concrete child classes from the VBC with the desired model implementation, adding them to its selection table
  - Defining a fromDictionary method, which is used during the run-time selection for construction from a dictionary
  - Selection of the model with a common interface (selector), using the model's name and a dictionary, which is fed to the fromDictionary classmethod of the required child-class
  - **Template of class inheritance structure** can be found in the templates/inheritance/ folder of the GitHub repository



- **BaseClass** – Virtual Base Class with implementation of run-time selection for handling model choice
- **Dictionary** – Ordered dictionary embedding useful I/O and access functionalities



# Input/output – Dictionary

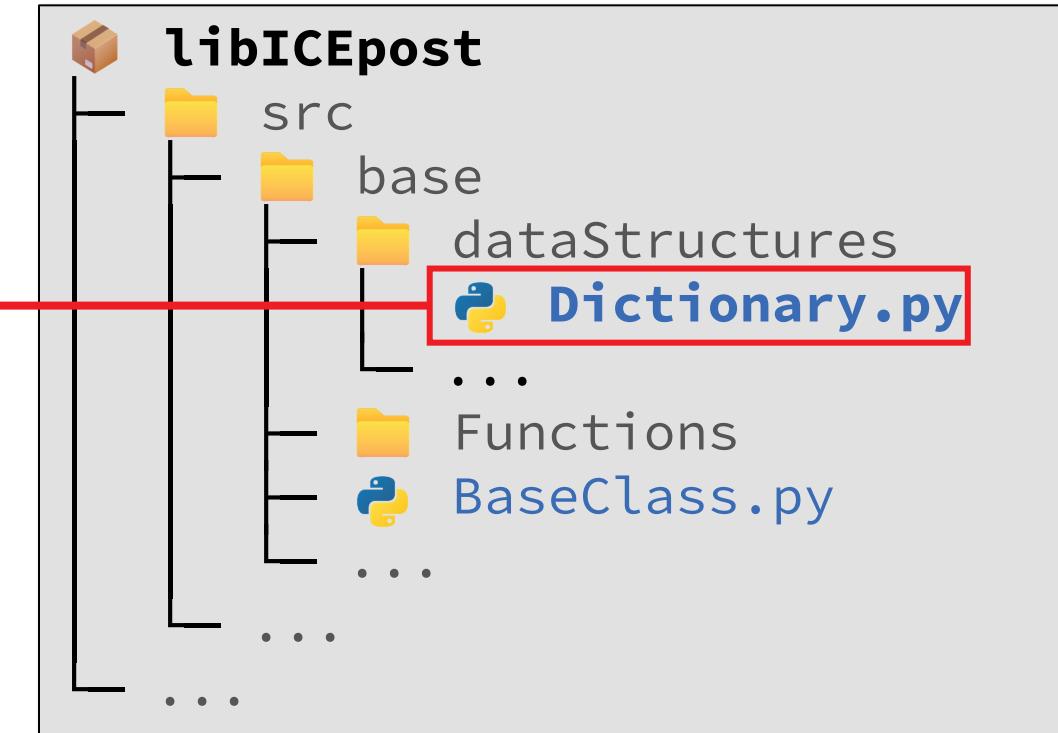
20

- **BaseClass** – Virtual Base Class with implementation of run-time selection for handling model choice
  - **Dictionary** – Ordered dictionary embedding useful I/O and access functionalities
    - fromFile: load information **from a python file**. The **file is executed** and all local variables are stored in the dictionary

```
1 #This is a string
2 myString = "a string"
3
4 # This is py
5 pi = 3.1415
```

user@localhost: ~

```
>>> from libICEpost.src.base.dataStructures.Dictionary import Dictionary
>>> dictionary = Dictionary.fromFile("./inputDictionary.py")
>>> print(dictionary)
Dictionary([('myString', 'a string'), ('pi', 3.1415)])
>>> dictionary.lookup("pi")
3.1415
>>> dictionary.lookupOrDefault("g", 9.81)
9.81
```

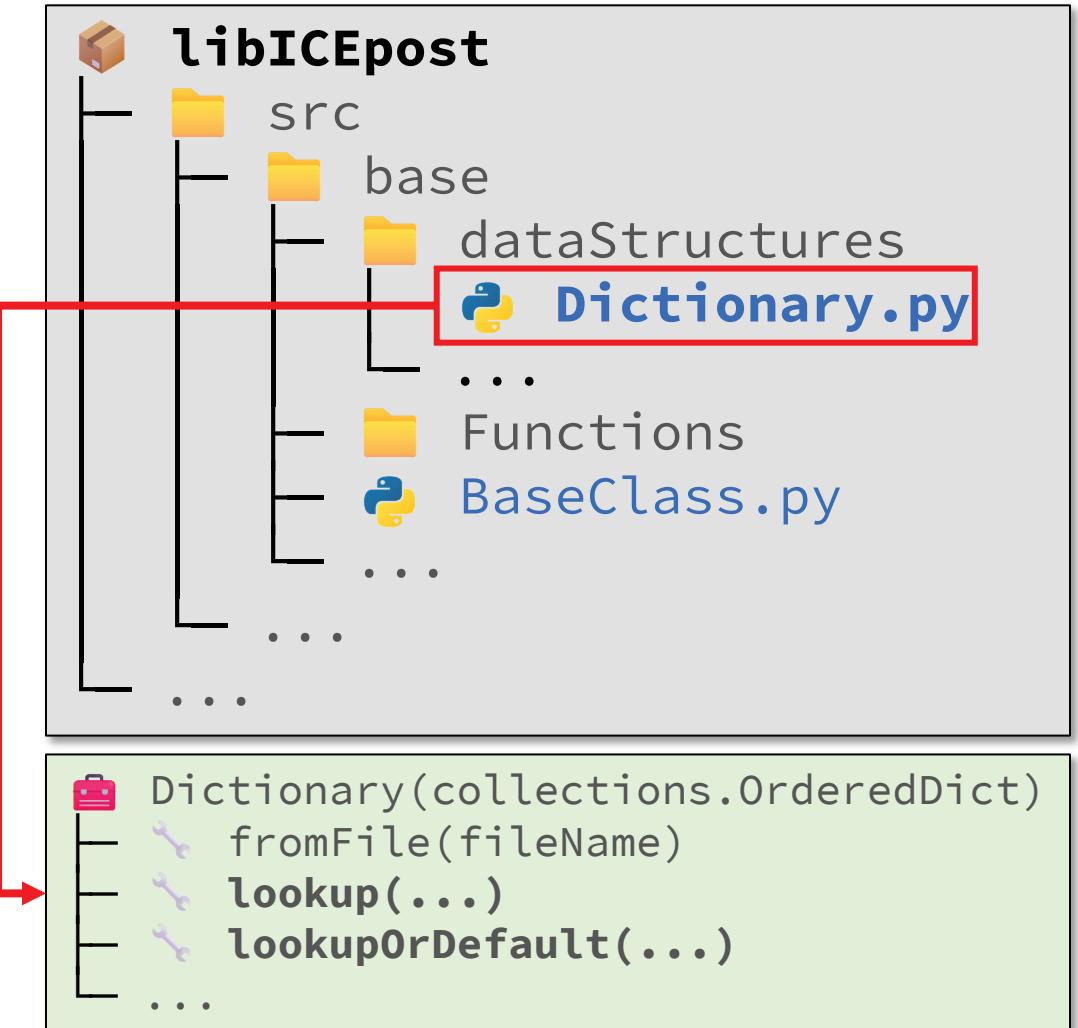


- Dictionary(collections.OrderedDict)
  - fromFile(fileName)
  - lookup(...)
  - lookupOrDefault(...)
  - ...

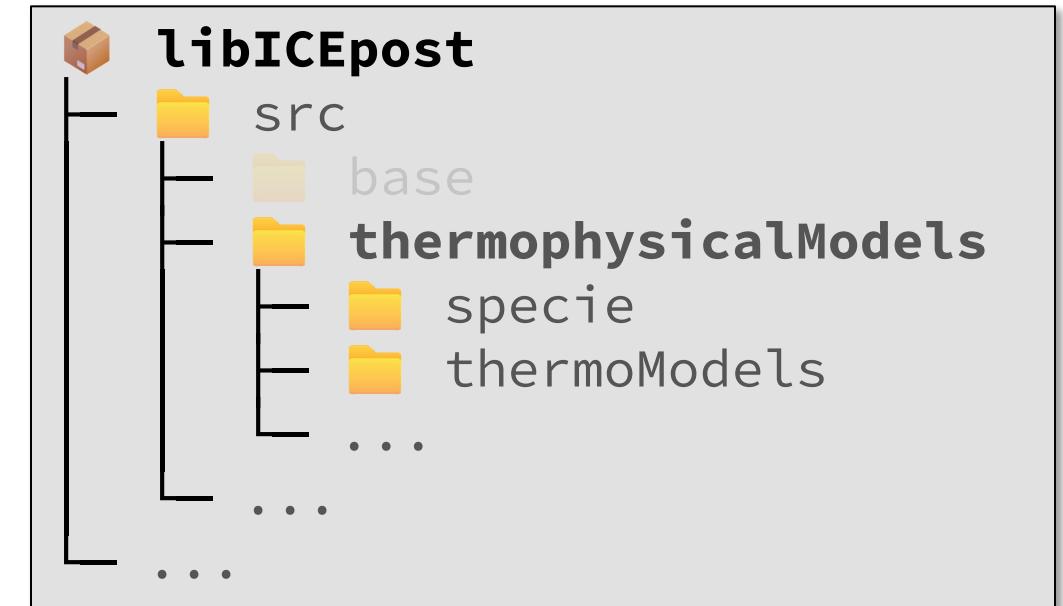
- **BaseClass** – Virtual Base Class with implementation of run-time selection for handling model choice
- **Dictionary** – Ordered dictionary embedding useful **I/O** and **access functionalities**
  - fromFile: load information from a python file. The file is executed and all local variables are stored in the dictionary
  - Convenient **access functions** consistent with OpenFOAM wording, with convenient type-checking and error handling

```
user@localhost: ~

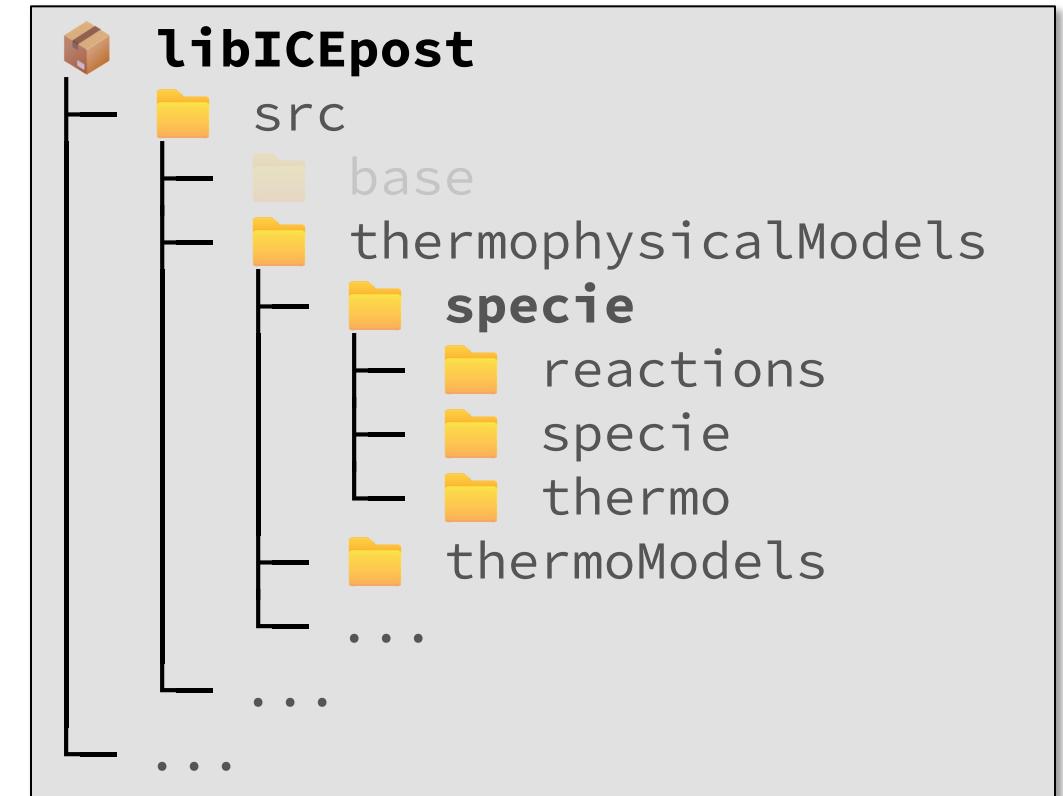
>>> from libICEpost.src.base.dataStructures.Dictionary import Dictionary
>>> dictionary = Dictionary.fromFile("./inputDictionary.py")
>>> print(dictionary)
Dictionary([('myString', 'a string'), ('pi', 3.1415)])
>>> dictionary.lookup("pi")
3.1415
>>> dictionary.lookupOrDefault("g", 9.81)
9.81
```



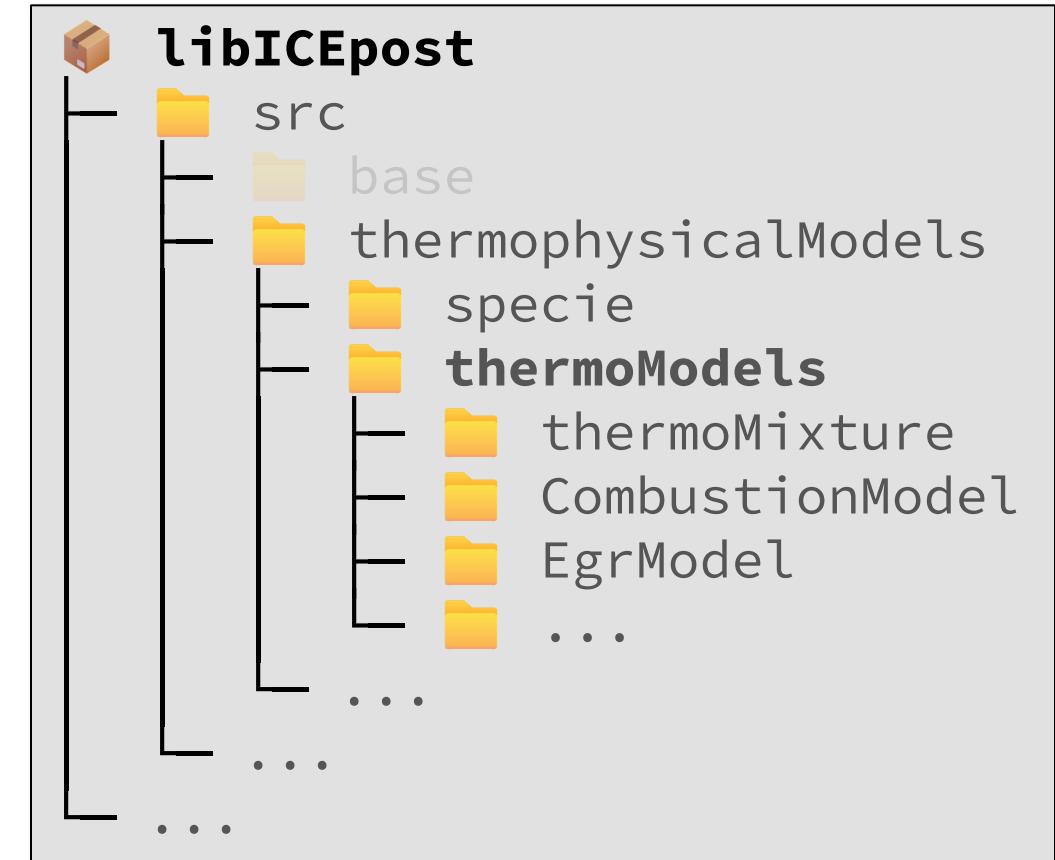
- Contains classes and functions for **modeling of chemistry and thermophysical properties** of mixtures



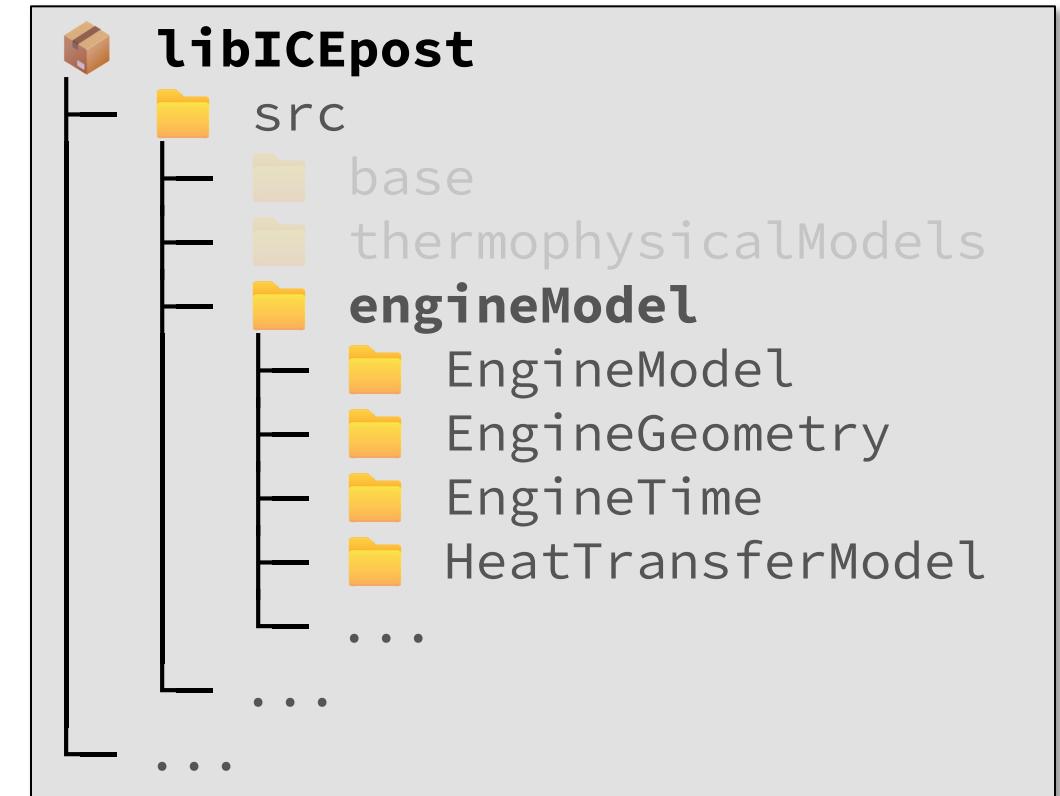
- Contains classes and functions for **modeling of chemistry and thermophysical properties** of mixtures
- **Main subpackages:**
  - **specie** – base **classes** and **function** for handling chemistry, mixtures, thermodynamic properties, and equations of state



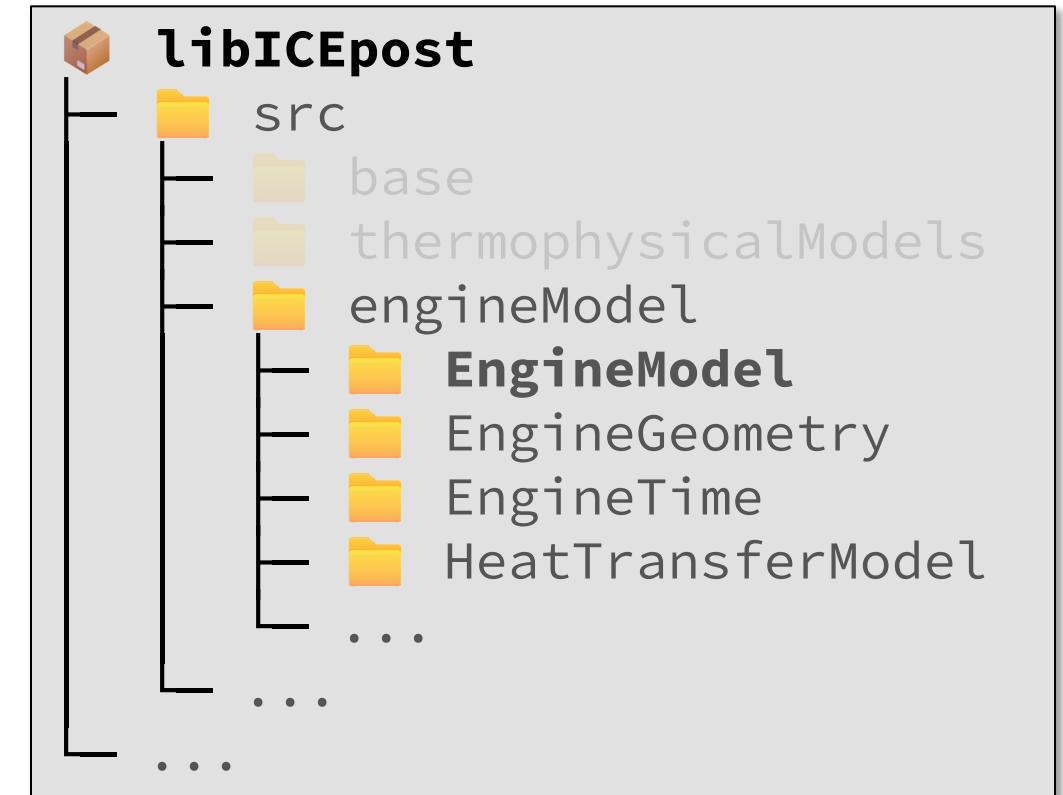
- Contains classes and functions for **modeling of chemistry and thermophysical properties** of mixtures
- **Main subpackages:**
  - specie – base classes and function for handling chemistry, mixtures, thermodynamic properties, and equations of state
  - **thermoModels** – chemical and thermodynamic modeling of complex systems:
    - Thermodynamic properties of mixtures (mixing rules)
    - Combustion models (2-zone model, etc.)
    - Computation of exhaust-gas composition
    - Etc...



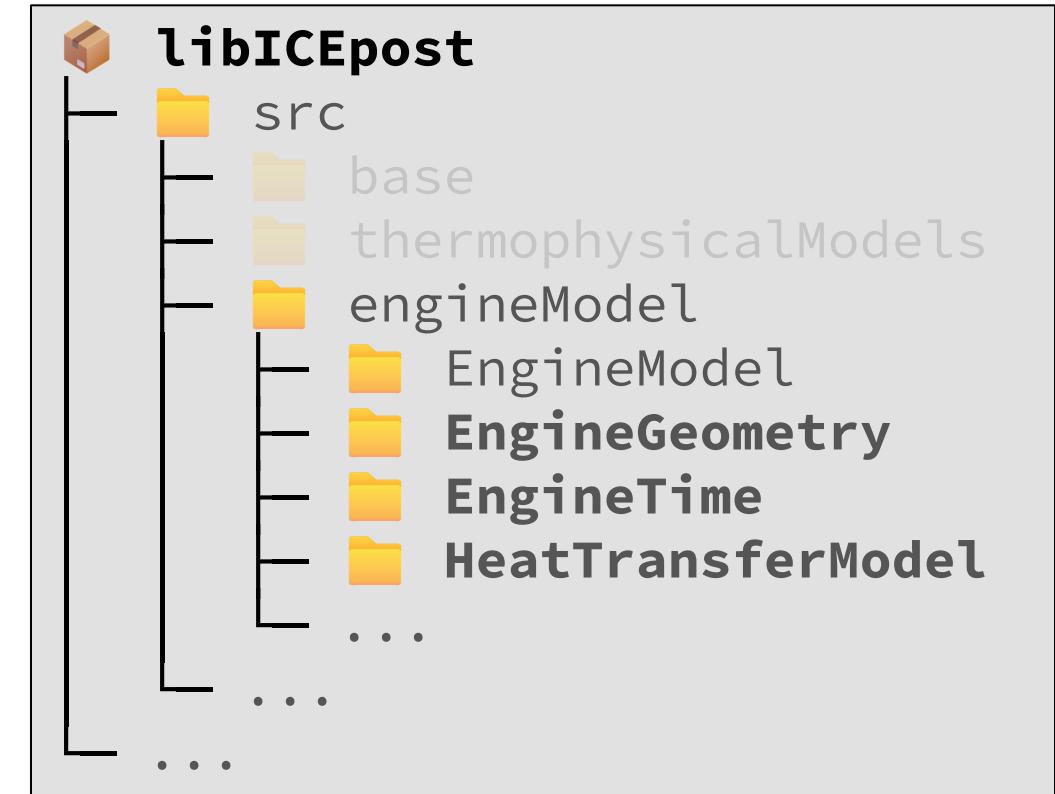
- Classes for modeling of **internal combustion engines** (ICE)
- **Processing results** of measurements/simulations and extrapolate additional information on the **development of the combustion process**



- Classes for modeling of **internal combustion engines** (ICE)
- **Processing results** of measurements/simulations and extrapolate additional information on the **development of the combustion process**
- **Main subpackages:**
  - **EngineModel** – defines the classes for **modeling of an engine configuration**, coupling all the sub-models

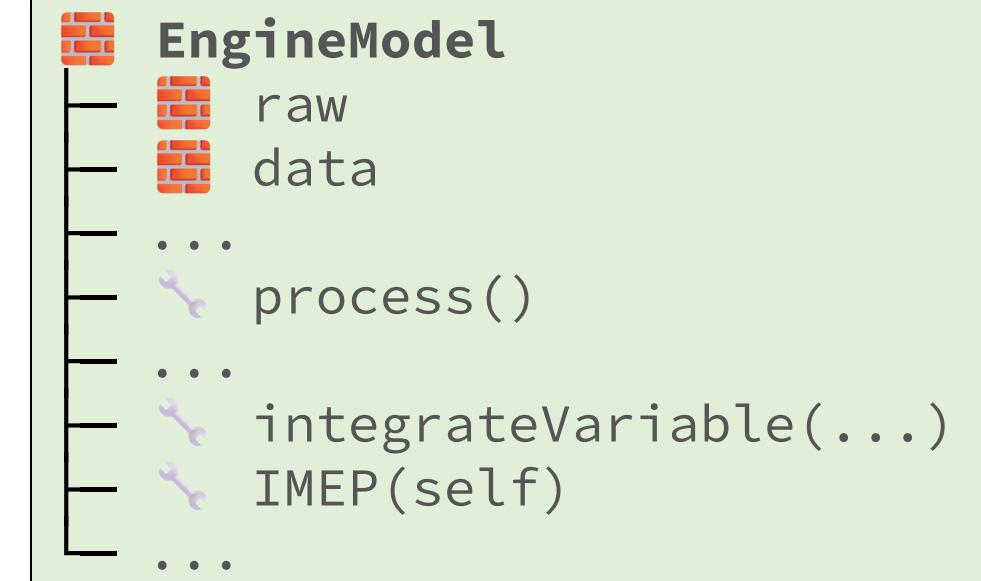


- Classes for modeling of **internal combustion engines** (ICE)
- **Processing results** of measurements/simulations and extrapolate additional information on the **development of the combustion process**
- **Main subpackages:**
  - **EngineModel** – defines the classes for modeling of an engine configuration, coupling all the sub-models
  - Multiple **sub-models** for handling specific aspects that may vary among different engine configurations, like:
    - EngineTime: handles the temporal evolution
    - EngineGeometry: geometrical features of the engine
    - HeatTransferModel: computes heat-transfer at walls

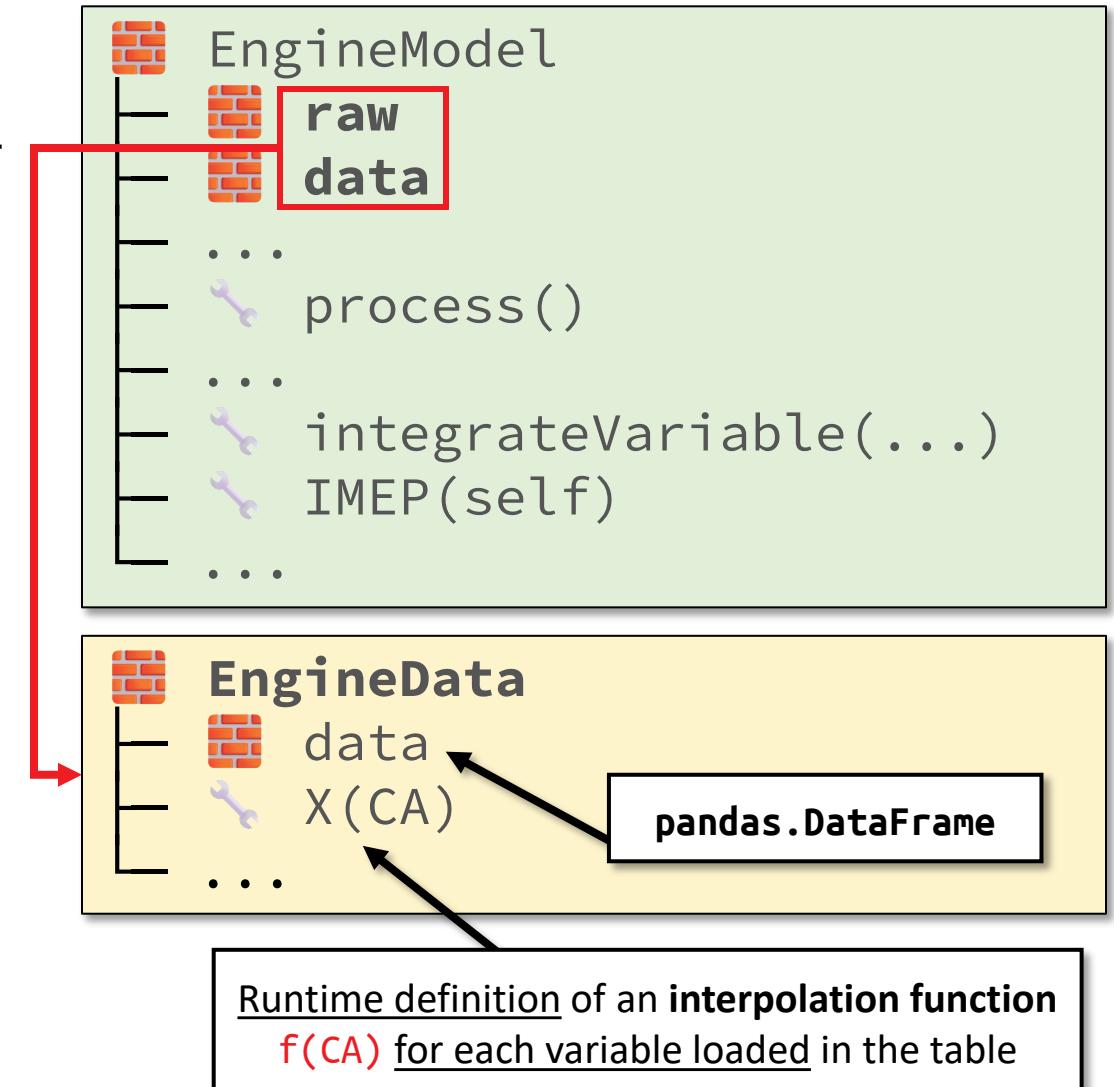


# EngineModel class

- **Base class for modeling ICE, implementing the generic features and coupling all the sub-models**

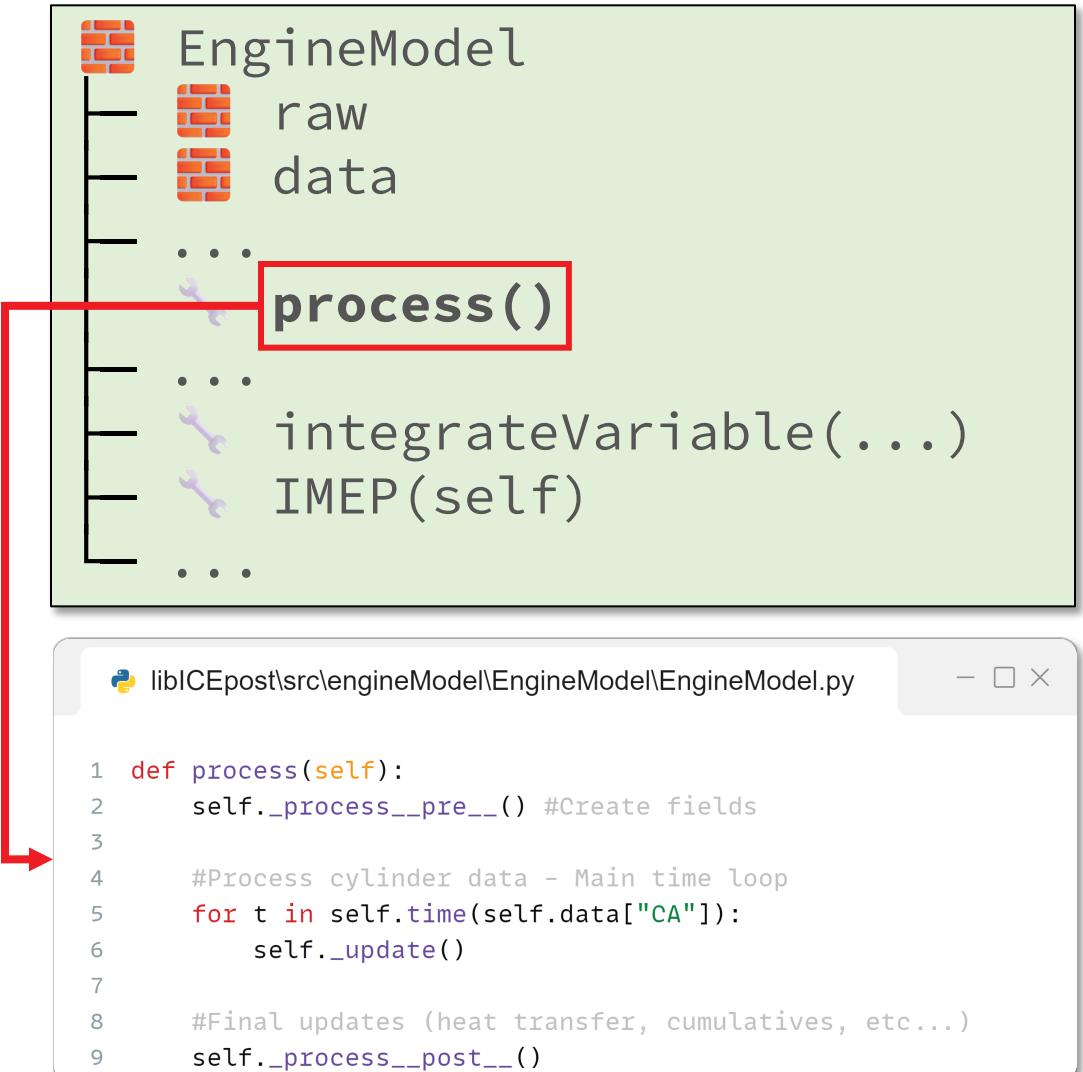


- Base class for modeling ICE, implementing the generic features and coupling all the sub-models
- Structure of an **object** of this **class**:
  - Two **data-structures** which contain the original (`raw`) and processed (`data`) data. They are instances of the EngineData class, a wrapper of a pandas.DataFrame

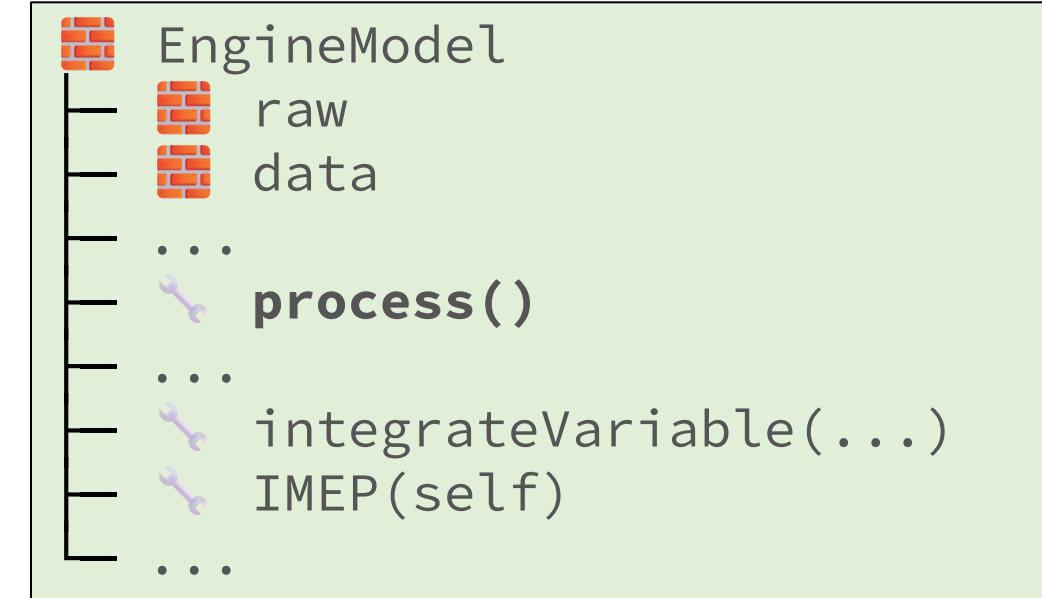


# EngineModel class

- Base class for modeling ICE, implementing the generic features and coupling all the sub-models
- Structure of an **object** of this **class**:
  - Two data-structures which contain the original (`raw`) and processed (`data`) data. They are instances of the `EngineData` class, a wrapper of a `pandas.DataFrame`
  - **`process()`** – The **main method** which processes the data and estimates the thermodynamic evolution of the system.



- Base class for modeling ICE, implementing the generic features and coupling all the sub-models
- Structure of an **object of this class**:
  - Two data-structures which contain the original (raw) and processed (data) data. They are instances of the EngineData class, a wrapper of a pandas.DataFrame
  - **process()** – The main method which processes the data and estimates the thermodynamic evolution of the system. Computes the **energy and mass balance at the cylinder** to estimate the apparent heat release rate (AHRR) and the rate of heat lease (ROHR) through the modelled wall heat flux ( $\dot{Q}_{walls}$ )



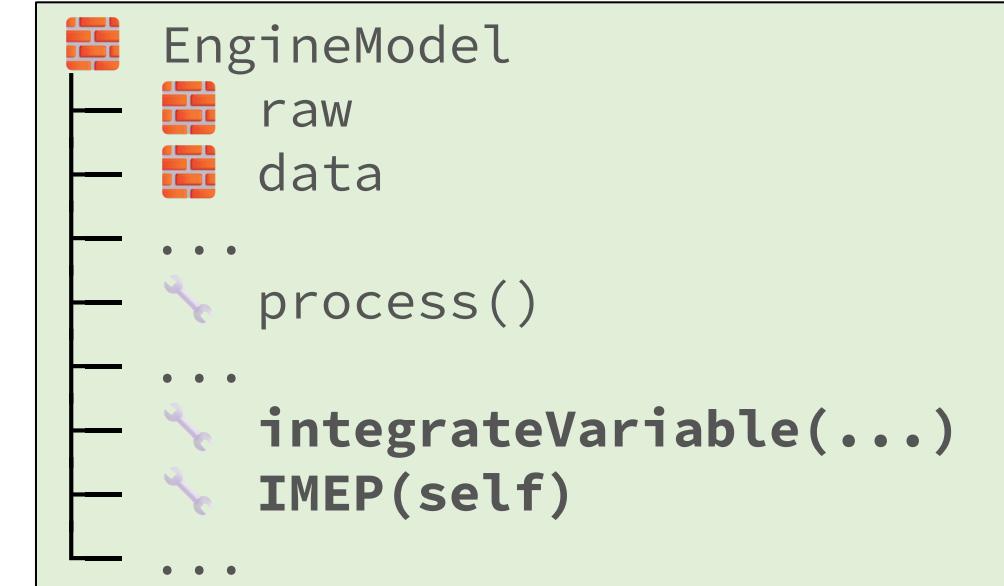
$$AHRR = \frac{dU_s}{d\theta} - \frac{dm_f}{d\theta} h_s - p \frac{dV}{d\theta}$$

Energy balance

$$ROHR = AHRR - \dot{Q}_{walls}$$

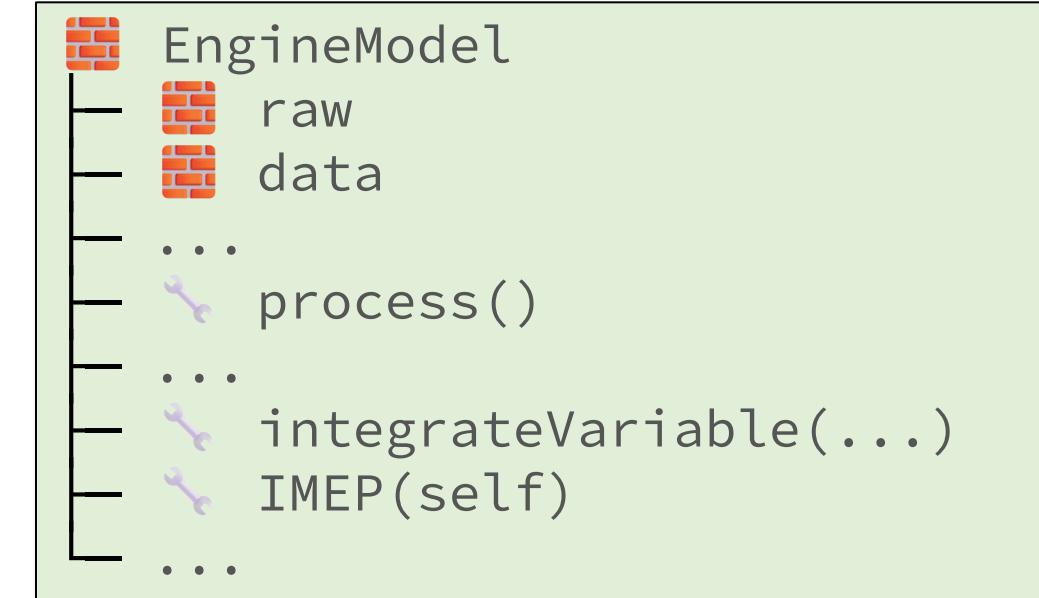
# EngineModel class

- **Base class for modeling ICE, implementing the generic features and coupling all the sub-models**
- **Structure of an object of this class:**
  - Two data-structures which contain the original (raw) and processed (data) data. They are instances of the EngineData class, a wrapper of a pandas.DataFrame
  - **process()** – The main method which processes the data and estimates the thermodynamic evolution of the system. Computes the energy and mass balance at the cylinder to estimate the apparent heat release rate (AHRR) and the rate of heat lease (ROHR) through the modelled wall heat flux ( $\dot{Q}_{walls}$ )
  - **Auxiliary methods** to extrapolate additional information from the processed data (integrating quantities, cumulative functions, work, etc.)

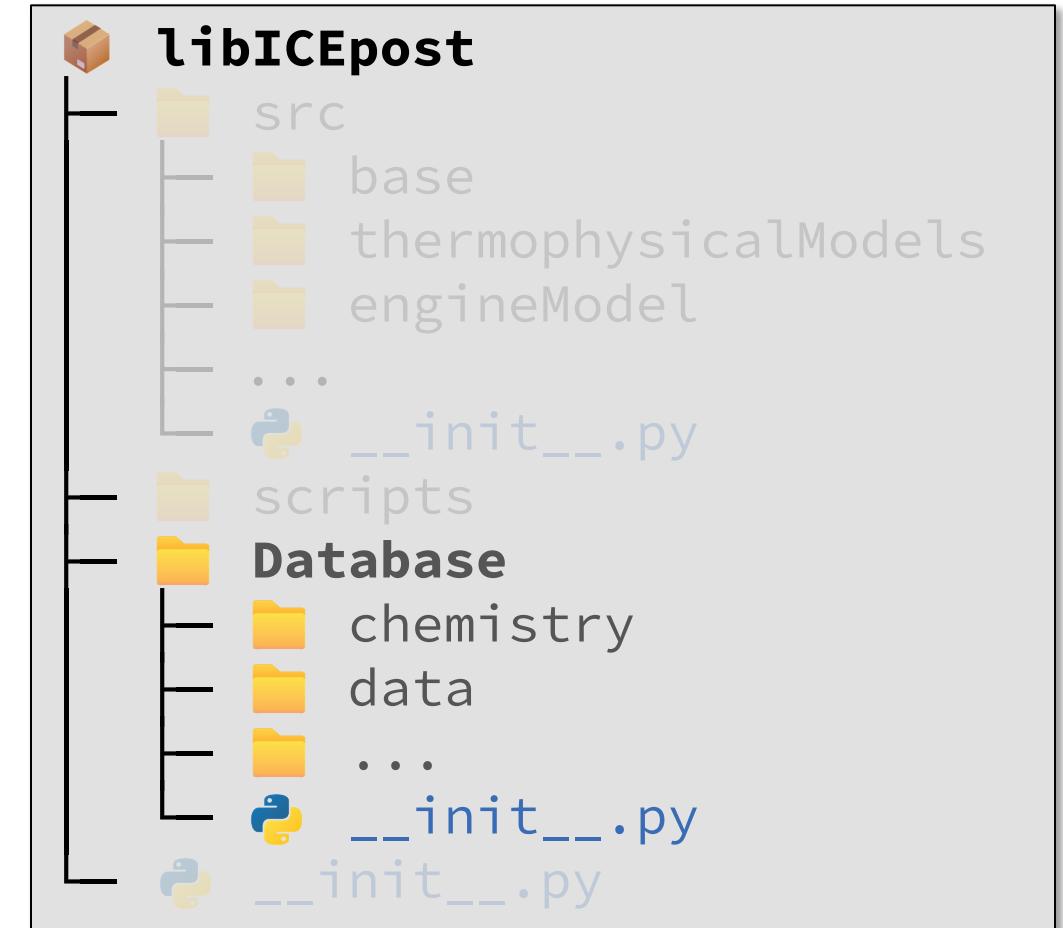
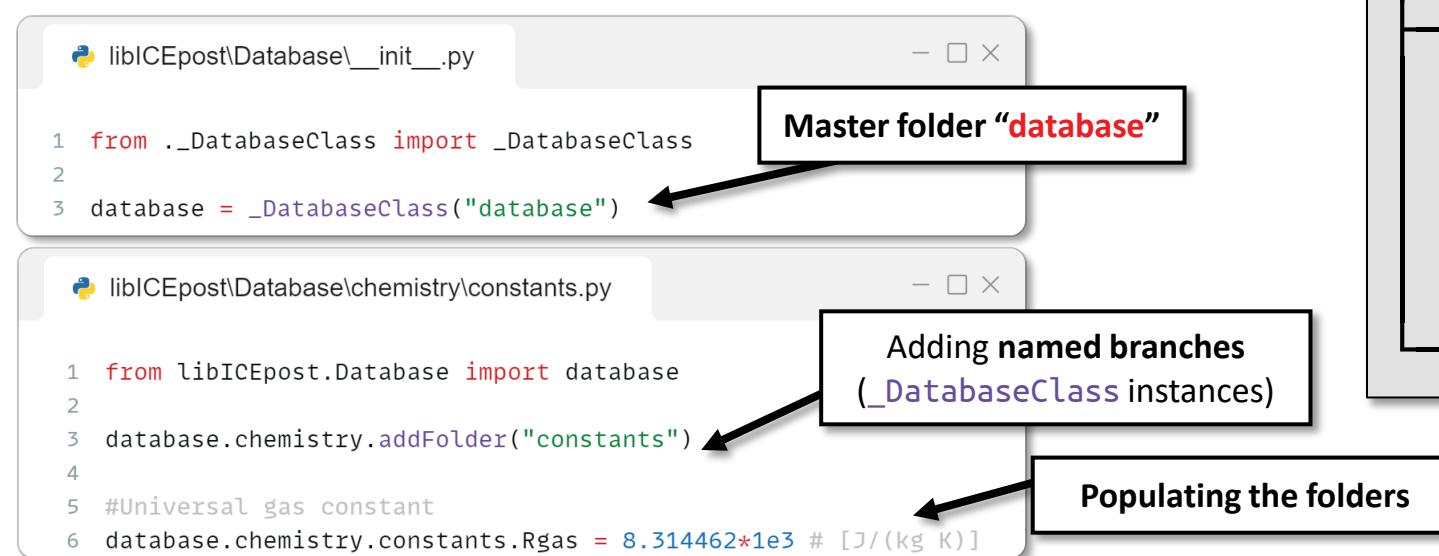


# EngineModel class

- **Base class for modeling ICE, implementing the generic features and coupling all the sub-models**
- **Structure of an object of this class:**
  - Two data-structures which contain the original (`raw`) and processed (`data`) data. They are instances of the `EngineData` class, a wrapper of a `pandas.DataFrame`
  - **`process()`** – The main method which processes the data and estimates the thermodynamic evolution of the system. Computes the energy and mass balance at the cylinder to estimate the apparent heat release rate (AHRR) and the rate of heat lease (ROHR) through the modelled wall heat flux ( $\dot{Q}_{walls}$ )
  - **Auxiliary methods** to extrapolate additional information from the processed data (integrating quantities, cumulative functions, work, etc.)
- References can be found in the docs/ folder

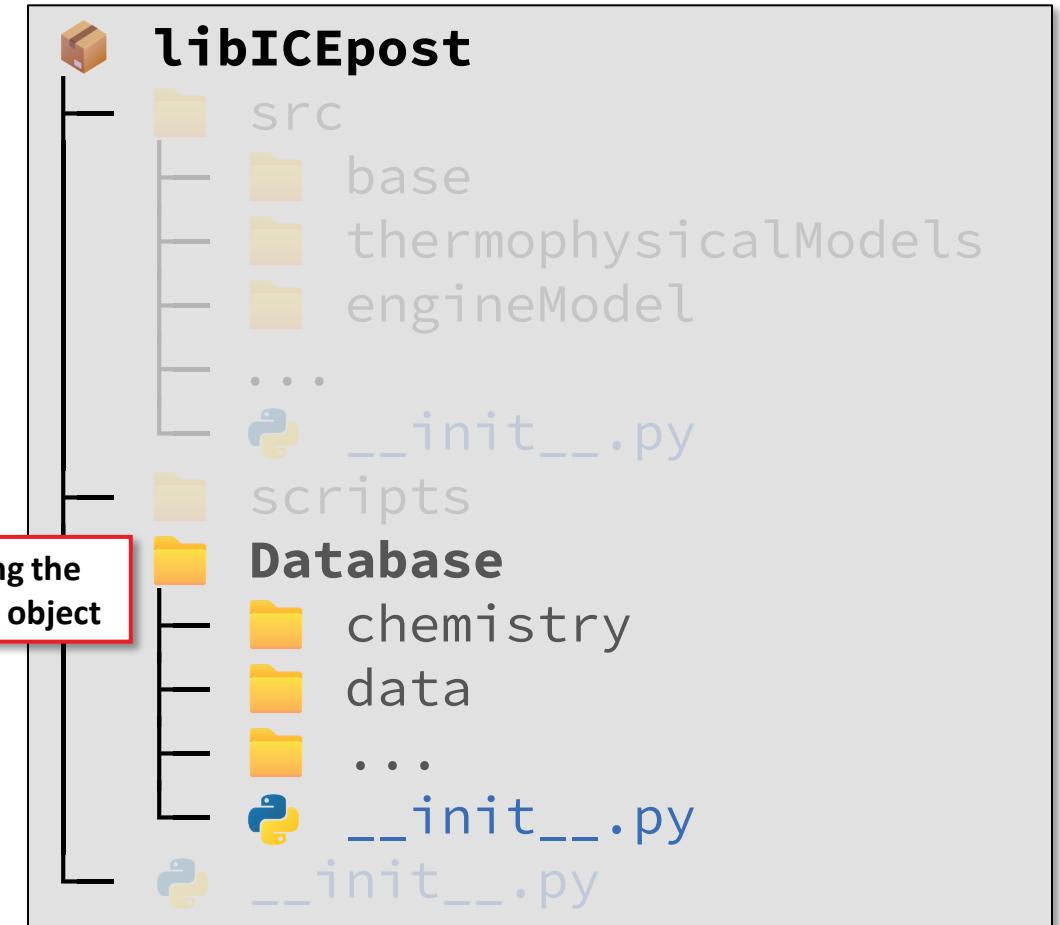


- A built-in database with useful data (periodic table, molecules, fuels, thermophysical properties, etc.)
- Contains the **variable database**: a container with a folder-like structure, which is dynamically populated loading the library sub-packages
- It accepts both dot (.) and key ([str]) access



- A built-in database with useful data (periodic table, molecules, fuels, thermophysical properties, etc.)
- Contains the **variable database**: a container with a folder-like structure, which is dynamically populated loading the library sub-packages
- It accepts both dot (.) and key ([str]) access

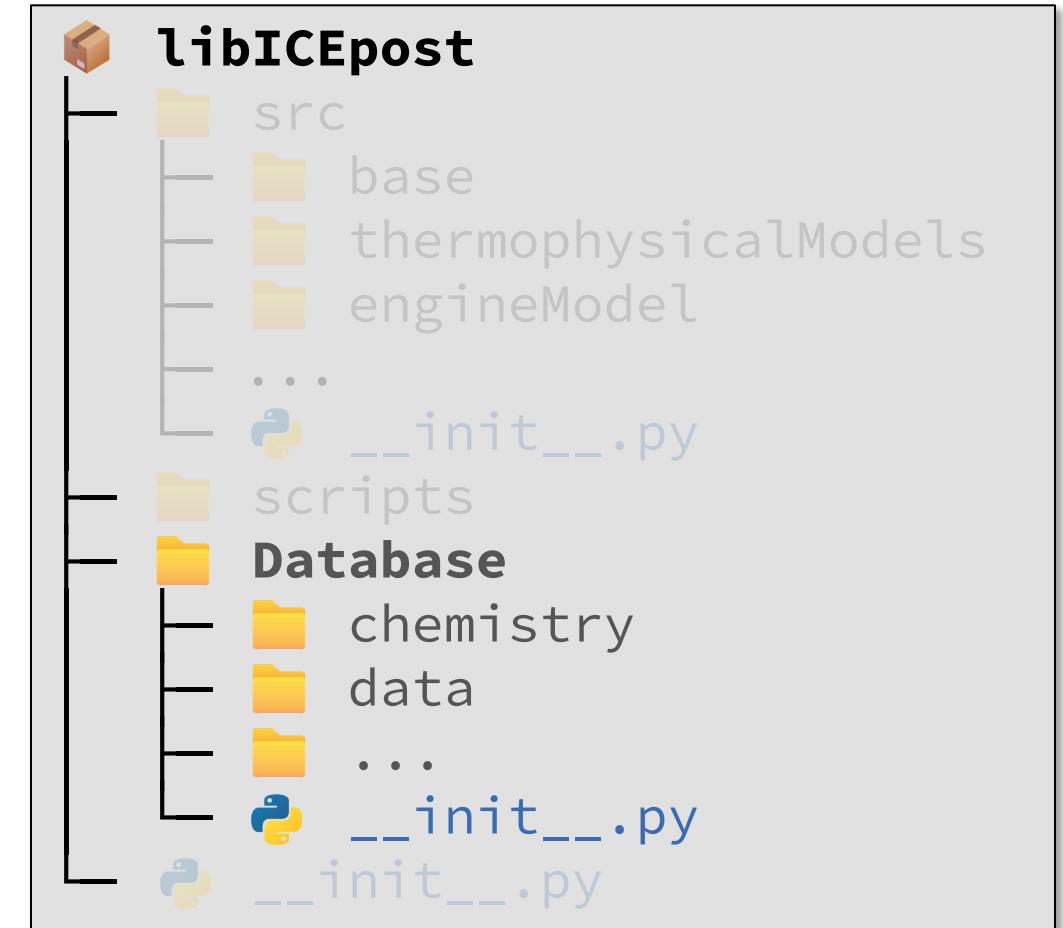
```
user@localhost: ~
Loading mixtures
in the database
>>> from libICEpost.Database.chemistry.specie.Mixtures import database
Importing the
database object
```



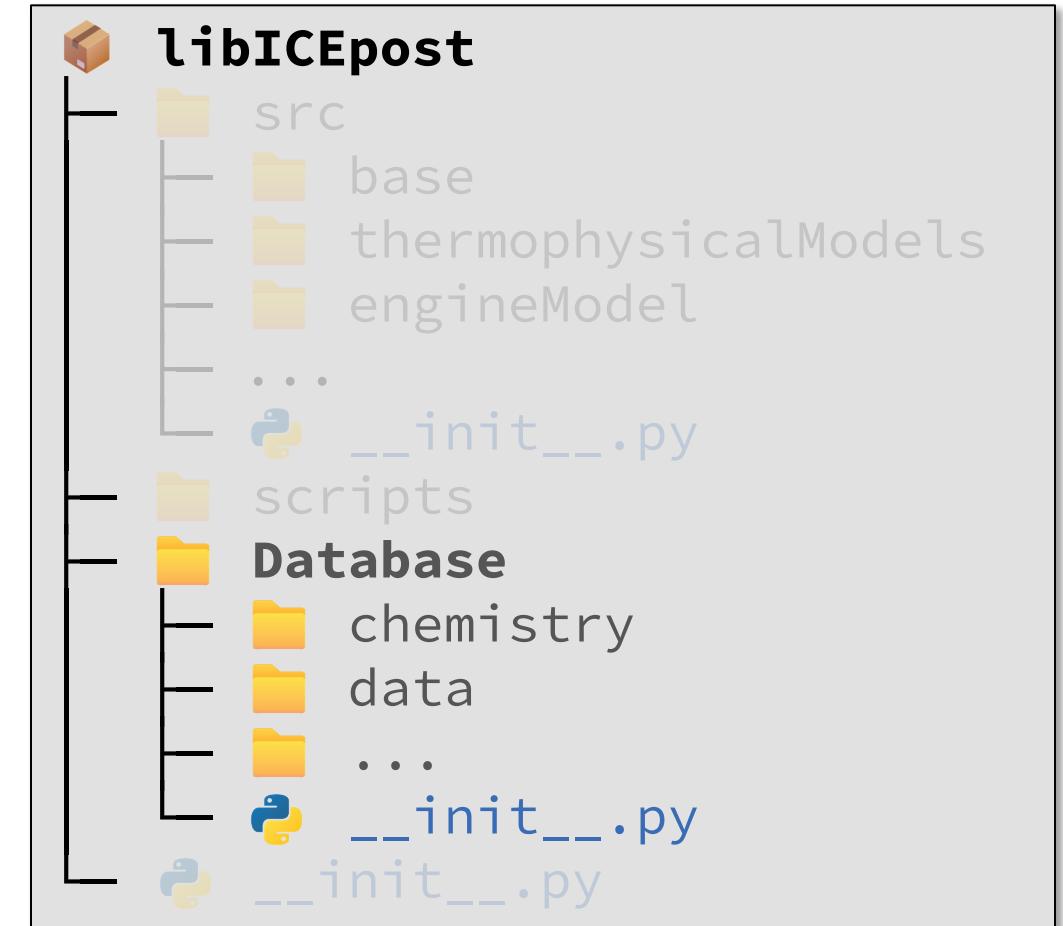
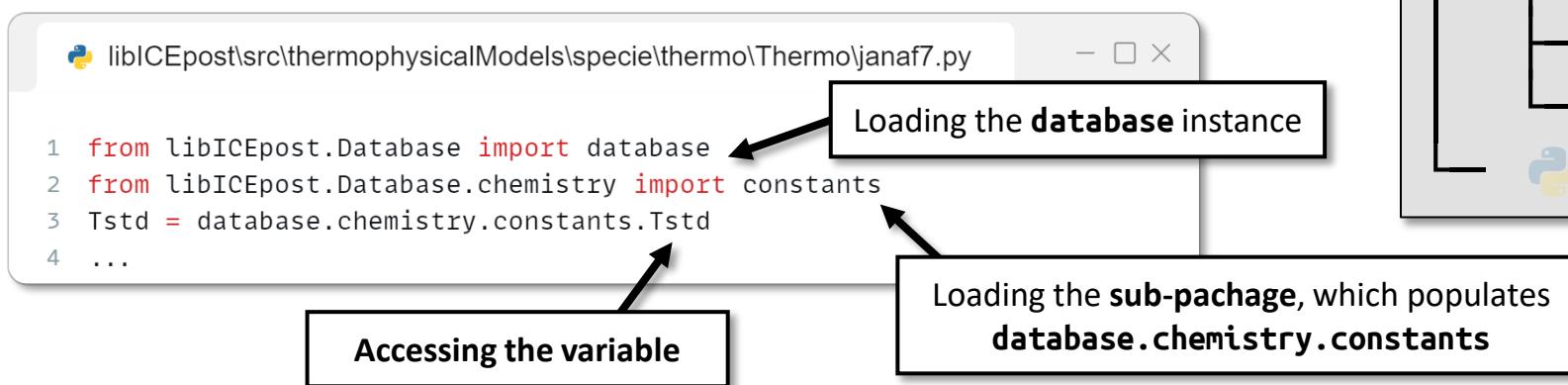
- A built-in database with useful data (periodic table, molecules, fuels, thermophysical properties, etc.)
- Contains the **variable database**: a container with a folder-like structure, which is dynamically populated loading the library sub-packages
- It accepts both dot (.) and key ([str]) access

```
user@localhost ~ % Name of the folder
>>> from libICEpost.Database import Database
>>> database
Database(database)[1] Number of elements in the folder
|- chemistry: Database(chemistry)[2]
| | |- specie: Database(specie)[4]
| | | |- periodicTable: Database(periodicTable)[118]
| | | | |- H: {'name': 'H', 'mass': 1.008}
| | | | |- He: {'name': 'He', 'mass': 4.002602}
| | | | # ...
| | | |- Molecules: Database(Molecules)[11]
| | | | |- CO2: {'name': 'CO2', 'mass': 44.009, 'atoms': [...]}

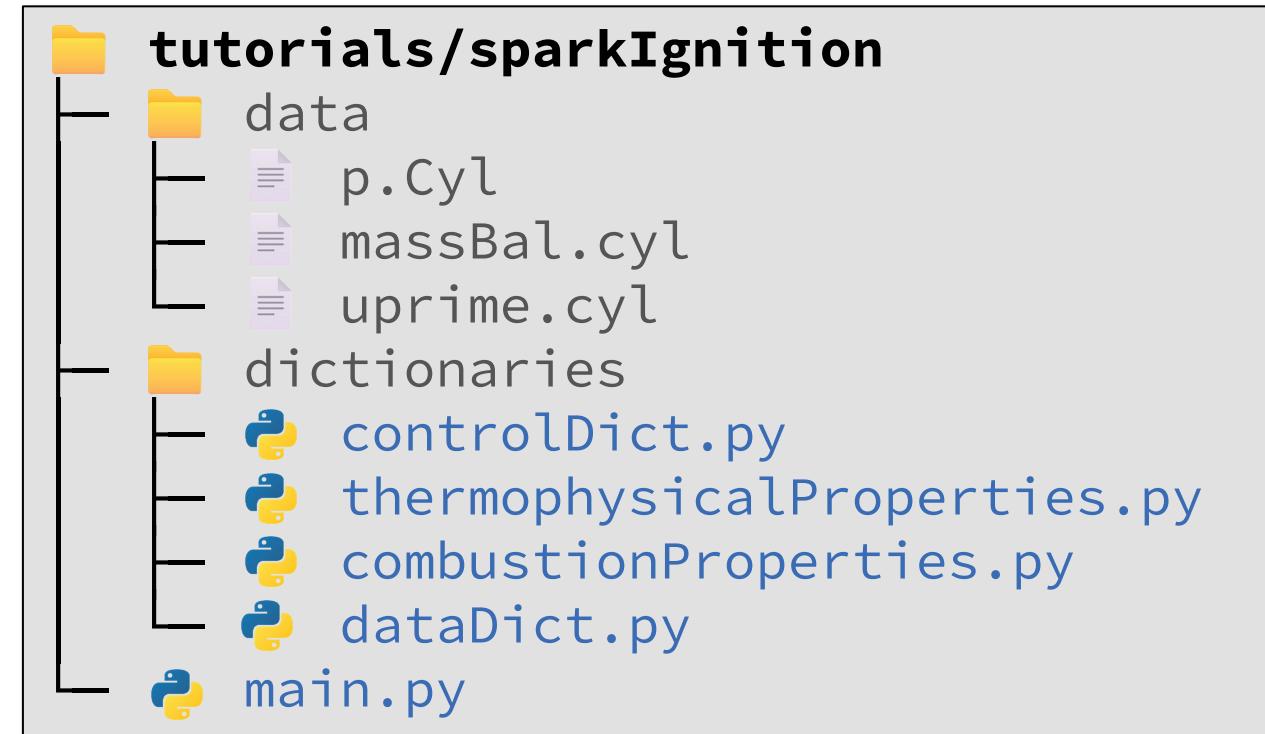
# ...
```



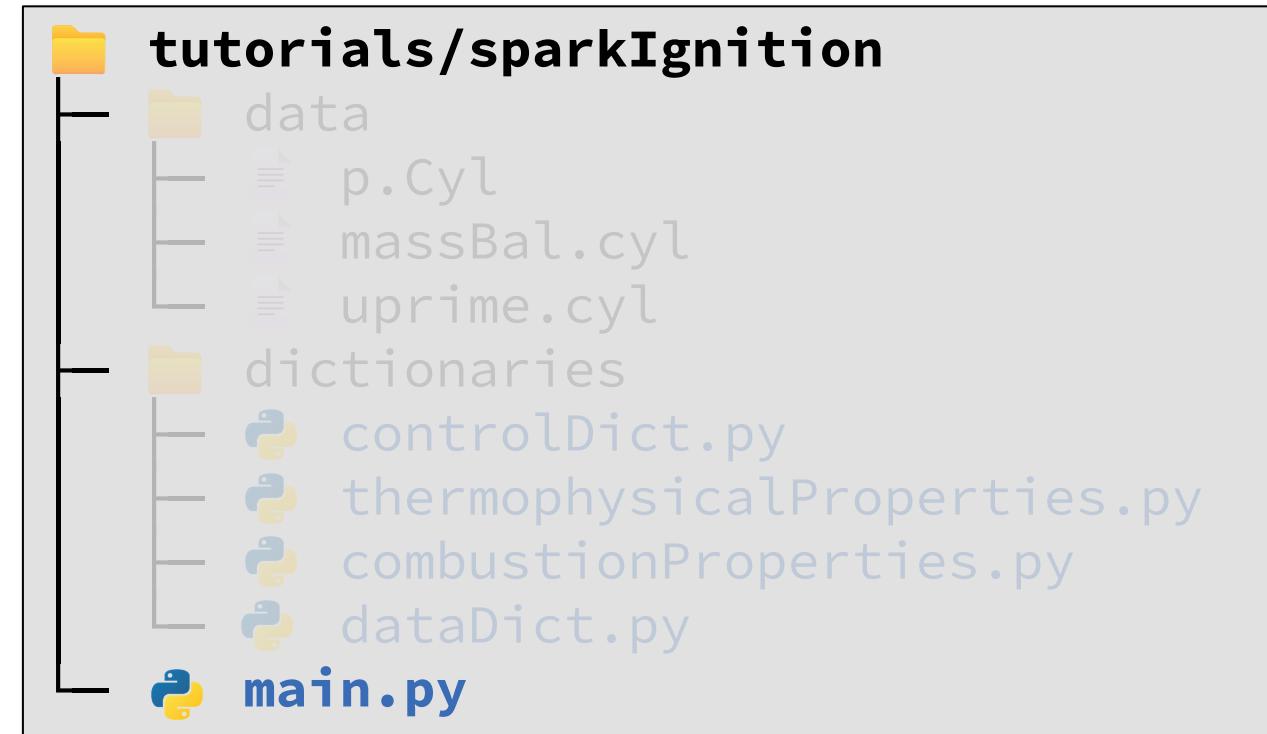
- A built-in database with useful data (periodic table, molecules, fuels, thermophysical properties, etc.)
- Contains the **variable database**: a container with a folder-like structure, which is dynamically populated loading the library sub-packages
- It accepts both dot (.) and key ([str]) access



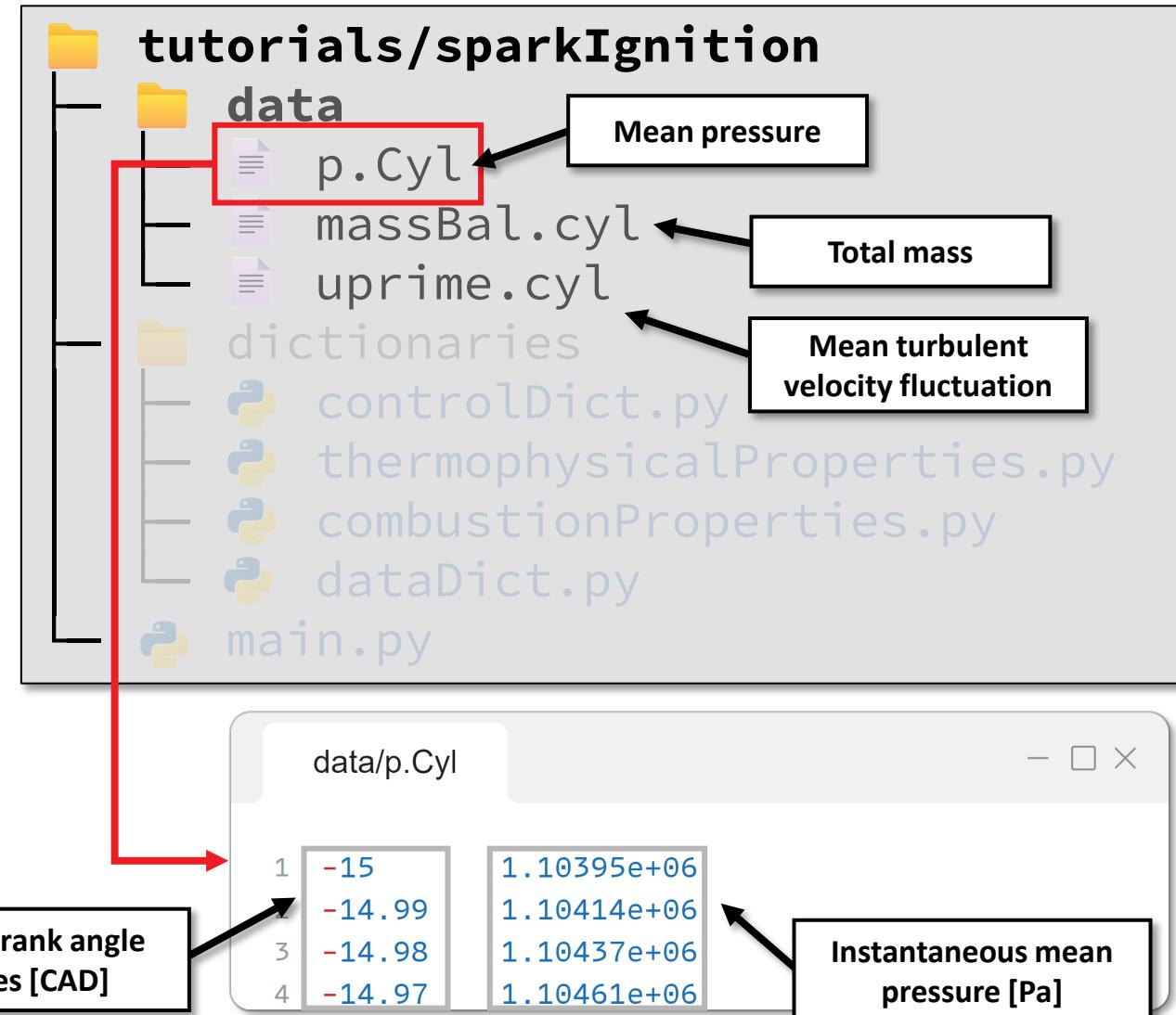
- **Tutorial case:** post-process the results of a CFD simulation of a spark-ignition engine to extrapolate the rate of heat release profile.
- The tutorial can be found in the **GitHub repository** at tutorials/sparkIgnition/



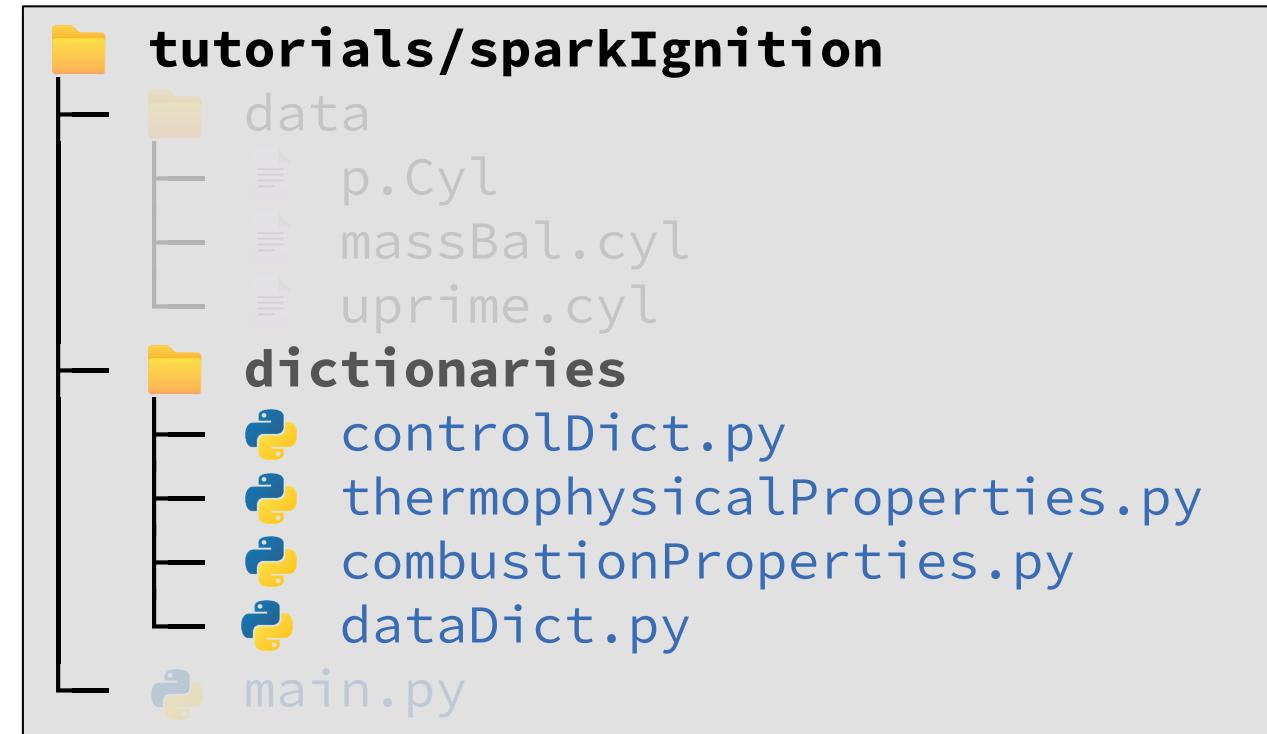
- Tutorial case: post-process the results of a CFD simulation of a spark-ignition engine to extrapolate the rate of heat release profile.
- The tutorial can be found in the **GitHub repository** at tutorials/sparkIgnition/
- **Content of the tutorial:**
  - **main.py** – The python script to be executed



- Tutorial case: post-process the results of a CFD simulation of a spark-ignition engine to extrapolate the rate of heat release profile.
- The tutorial can be found in the GitHub repository at tutorials/sparkIgnition/
- **Content of the tutorial:**
  - main.py – The python script to be executed
  - **data** – Contains **results of the CFD simulations**, i.e., in-cylinder traces of integral or average quantities



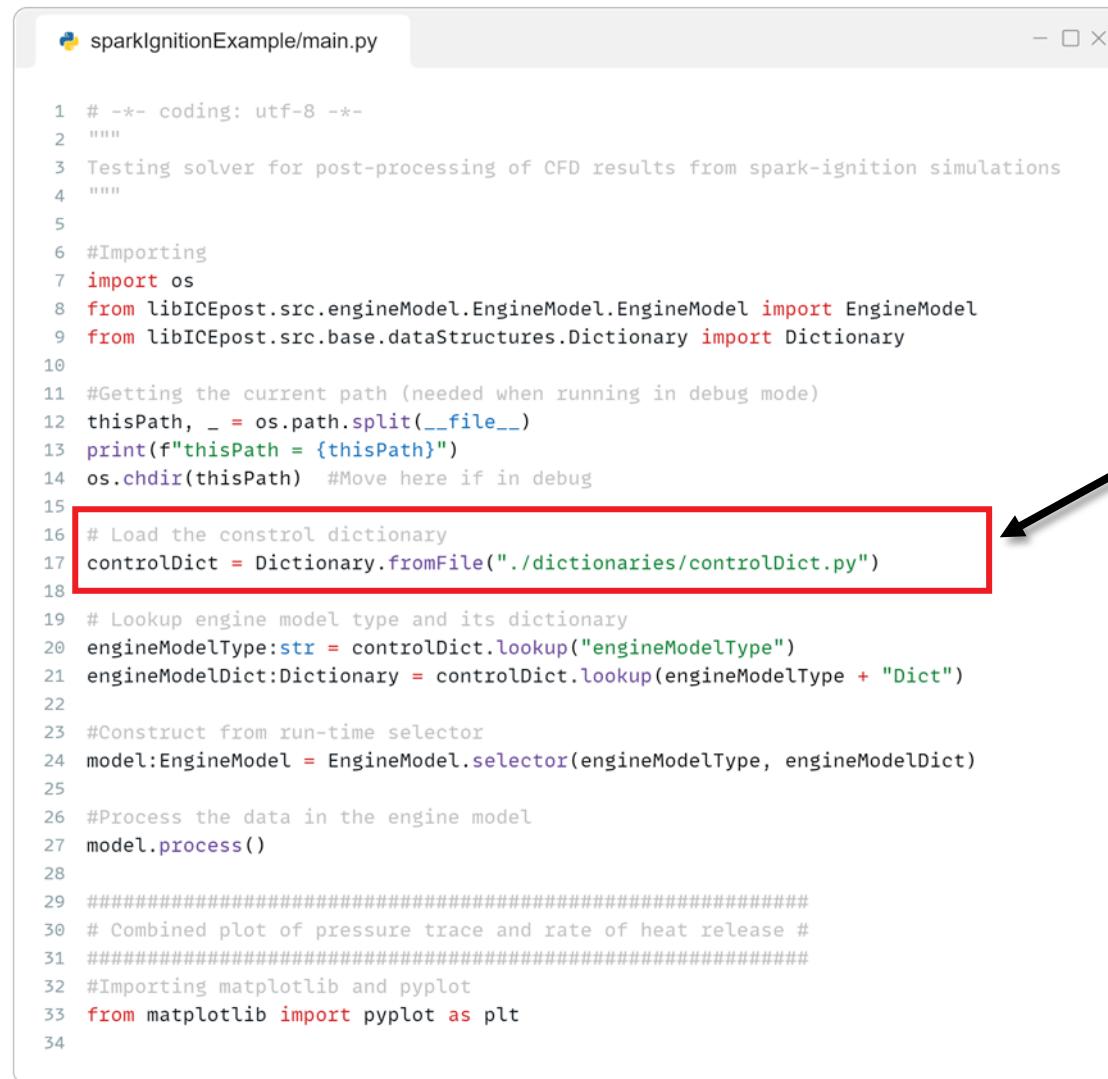
- Tutorial case: post-process the results of a CFD simulation of a spark-ignition engine to extrapolate the rate of heat release profile.
- The tutorial can be found in the **GitHub repository** at tutorials/sparkIgnition/
- **Content of the tutorial:**
  - main.py – The python script to be executed
  - data – Contains results of the CFD simulations, i.e., in-cylinder traces of integral or average quantities
  - **dictionaries** – Contains all the **dictionaries** that are used to construct the engine model



```
sparkIgnitionExample/main.py

1 # -*- coding: utf-8 -*-
2 """
3 Testing solver for post-processing of CFD results from spark-ignition simulations
4 """
5
6 #Importing
7 import os
8 from libICEpost.src.engineModel.EngineModel.EngineModel import EngineModel
9 from libICEpost.src.base.dataStructures.Dictionary import Dictionary
10
11 #Getting the current path (needed when running in debug mode)
12 thisPath, _ = os.path.split(__file__)
13 print(f"thisPath = {thisPath}")
14 os.chdir(thisPath) #Move here if in debug
15
16 # Load the control dictionary
17 controlDict = Dictionary.fromFile("./dictionaries/controlDict.py")
18
19 # Lookup engine model type and its dictionary
20 engineModelType:str = controlDict.lookup("engineModelType")
21 engineModelDict:Dictionary = controlDict.lookup(engineModelType + "Dict")
22
23 #Construct from run-time selector
24 model:EngineModel = EngineModel.selector(engineModelType, engineModelDict)
25
26 #Process the data in the engine model
27 model.process()
28
29 #####
30 # Combined plot of pressure trace and rate of heat release #
31 #####
32 #Importing matplotlib and pyplot
33 from matplotlib import pyplot as plt
34
```

Importing **EngineModel** and **Dictionary** from **libICEpost**



```
# -*- coding: utf-8 -*-
"""
Testing solver for post-processing of CFD results from spark-ignition simulations
"""

#Importing
import os
from libICEpost.src.engineModel.EngineModel import EngineModel
from libICEpost.src.base.dataStructures.Dictionary import Dictionary

#Getting the current path (needed when running in debug mode)
thisPath, _ = os.path.split(__file__)
print(f"thisPath = {thisPath}")
os.chdir(thisPath) #Move here if in debug

# Load the control dictionary
controlDict = Dictionary.fromFile("./dictionaries/controlDict.py")

# Lookup engine model type and its dictionary
engineModelType:str = controlDict.lookup("engineModelType")
engineModelDict:Dictionary = controlDict.lookup(engineModelType + "Dict")

#Construct from run-time selector
model:EngineModel = EngineModel.selector(engineModelType, engineModelDict)

#Process the data in the engine model
model.process()

#####
# Combined plot of pressure trace and rate of heat release #
#####

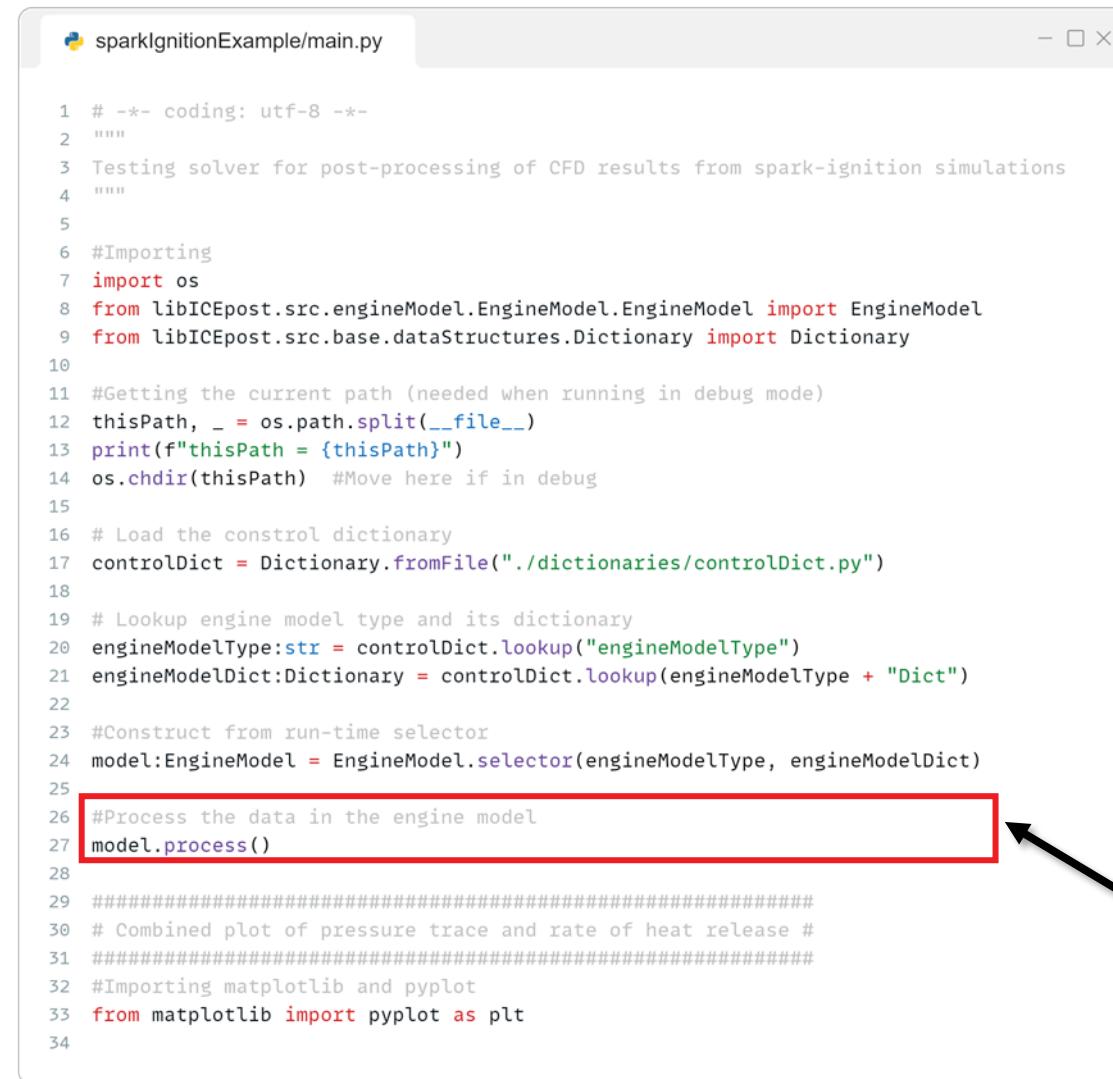
#Importing matplotlib and pyplot
from matplotlib import pyplot as plt
```

Importing the configuration of the engine model from controlDictionary.py file into a **Dictionary** entry with the fromFile **classmethod**

```
sparkIgnitionExample/main.py

1 # -*- coding: utf-8 -*-
2 """
3 Testing solver for post-processing of CFD results from spark-ignition simulations
4 """
5
6 #Importing
7 import os
8 from libICEpost.src.engineModel.EngineModel import EngineModel
9 from libICEpost.src.base.dataStructures.Dictionary import Dictionary
10
11 #Getting the current path (needed when running in debug mode)
12 thisPath, _ = os.path.split(__file__)
13 print(f"thisPath = {thisPath}")
14 os.chdir(thisPath) #Move here if in debug
15
16 # Load the control dictionary
17 controlDict = Dictionary.fromFile("./dictionaries/controlDict.py")
18
19 # Lookup engine model type and its dictionary
20 engineModelType:str = controlDict.lookup("engineModelType")
21 engineModelDict:Dictionary = controlDict.lookup(engineModelType + "Dict")
22
23 #Construct from run-time selector
24 model:EngineModel = EngineModel.selector(engineModelType, engineModelDict)
25
26 #Process the data in the engine model
27 model.process()
28
29 #####
30 # Combined plot of pressure trace and rate of heat release #
31 #####
32 #Importing matplotlib and pyplot
33 from matplotlib import pyplot as plt
34
```

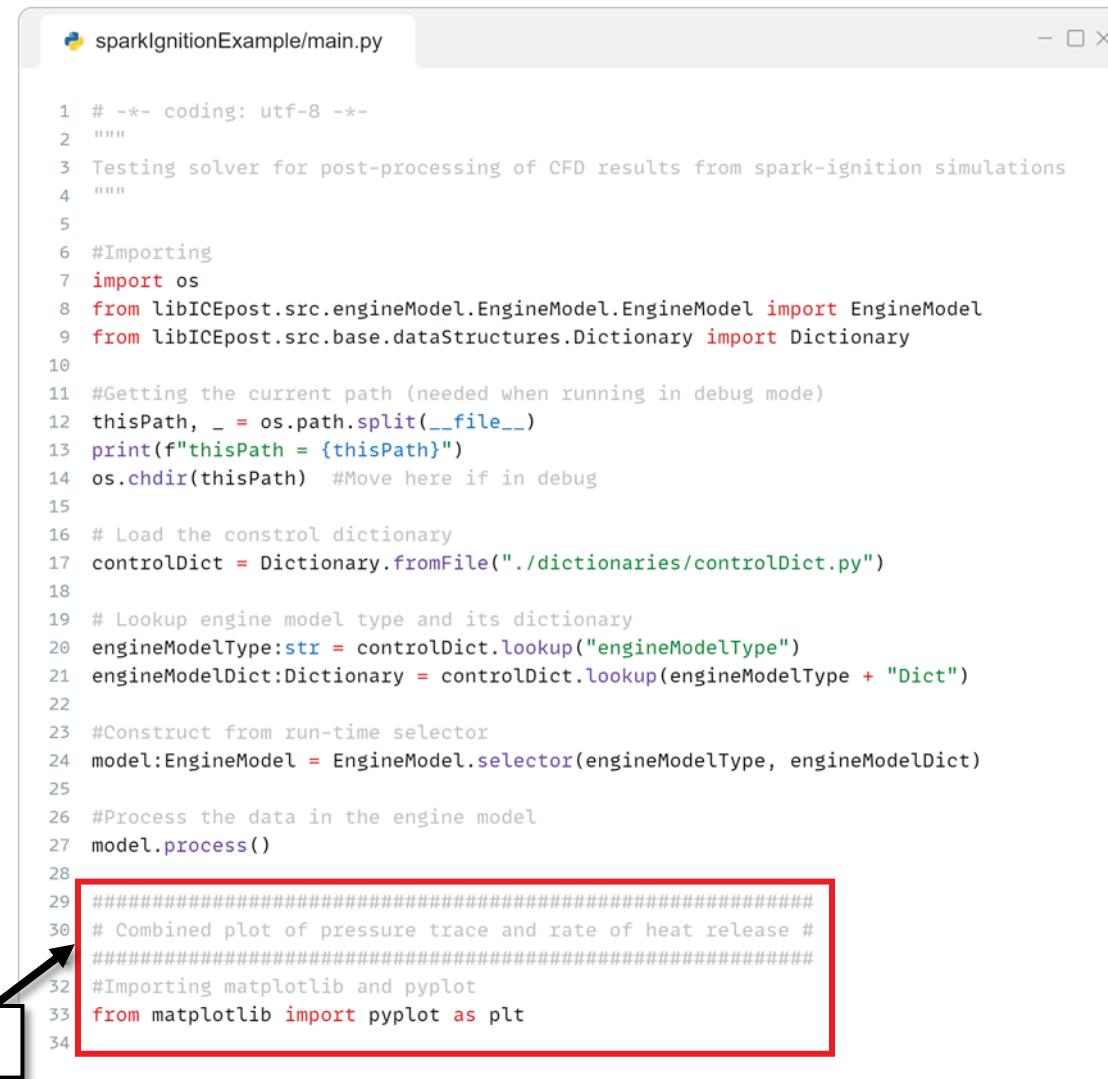
Lookup the EngineModel to use  
and its construction dictionary  
and then construct it through  
the **selector classmethod**



```
sparkIgnitionExample/main.py

1 # -*- coding: utf-8 -*-
2 """
3 Testing solver for post-processing of CFD results from spark-ignition simulations
4 """
5
6 #Importing
7 import os
8 from libICEpost.src.engineModel.EngineModel import EngineModel
9 from libICEpost.src.base.dataStructures.Dictionary import Dictionary
10
11 #Getting the current path (needed when running in debug mode)
12 thisPath, _ = os.path.split(__file__)
13 print(f"thisPath = {thisPath}")
14 os.chdir(thisPath) #Move here if in debug
15
16 # Load the control dictionary
17 controlDict = Dictionary.fromFile("./dictionaries/controlDict.py")
18
19 # Lookup engine model type and its dictionary
20 engineModelType:str = controlDict.lookup("engineModelType")
21 engineModelDict:Dictionary = controlDict.lookup(engineModelType + "Dict")
22
23 #Construct from run-time selector
24 model:EngineModel = EngineModel.selector(engineModelType, engineModelDict)
25
26 #Process the data in the engine model
27 model.process()
28
29 #####
30 # Combined plot of pressure trace and rate of heat release #
31 #####
32 #Importing matplotlib and pyplot
33 from matplotlib import pyplot as plt
34
```

Process the data loaded in the  
engine model



```
sparkIgnitionExample/main.py

1 # -*- coding: utf-8 -*-
2 """
3 Testing solver for post-processing of CFD results from spark-ignition simulations
4 """
5
6 #Importing
7 import os
8 from libICEpost.src.engineModel.EngineModel import EngineModel
9 from libICEpost.src.base.dataStructures.Dictionary import Dictionary
10
11 #Getting the current path (needed when running in debug mode)
12 thisPath, _ = os.path.split(__file__)
13 print(f"thisPath = {thisPath}")
14 os.chdir(thisPath) #Move here if in debug
15
16 # Load the control dictionary
17 controlDict = Dictionary.fromFile("./dictionaries/controlDict.py")
18
19 # Lookup engine model type and its dictionary
20 engineModelType:str = controlDict.lookup("engineModelType")
21 engineModelDict:Dictionary = controlDict.lookup(engineModelType + "Dict")
22
23 #Construct from run-time selector
24 model:EngineModel = EngineModel.selector(engineModelType, engineModelDict)
25
26 #Process the data in the engine model
27 model.process()
28
29 ######
30 # Combined plot of pressure trace and rate of heat release #
31 #####
32 #Importing matplotlib and pyplot
33 from matplotlib import pyplot as plt
34
```

Plotting results

# Setup files – controlDict.py

- This control dictionary contains the specific engine model to use and the construction dictionary of the corresponding class.



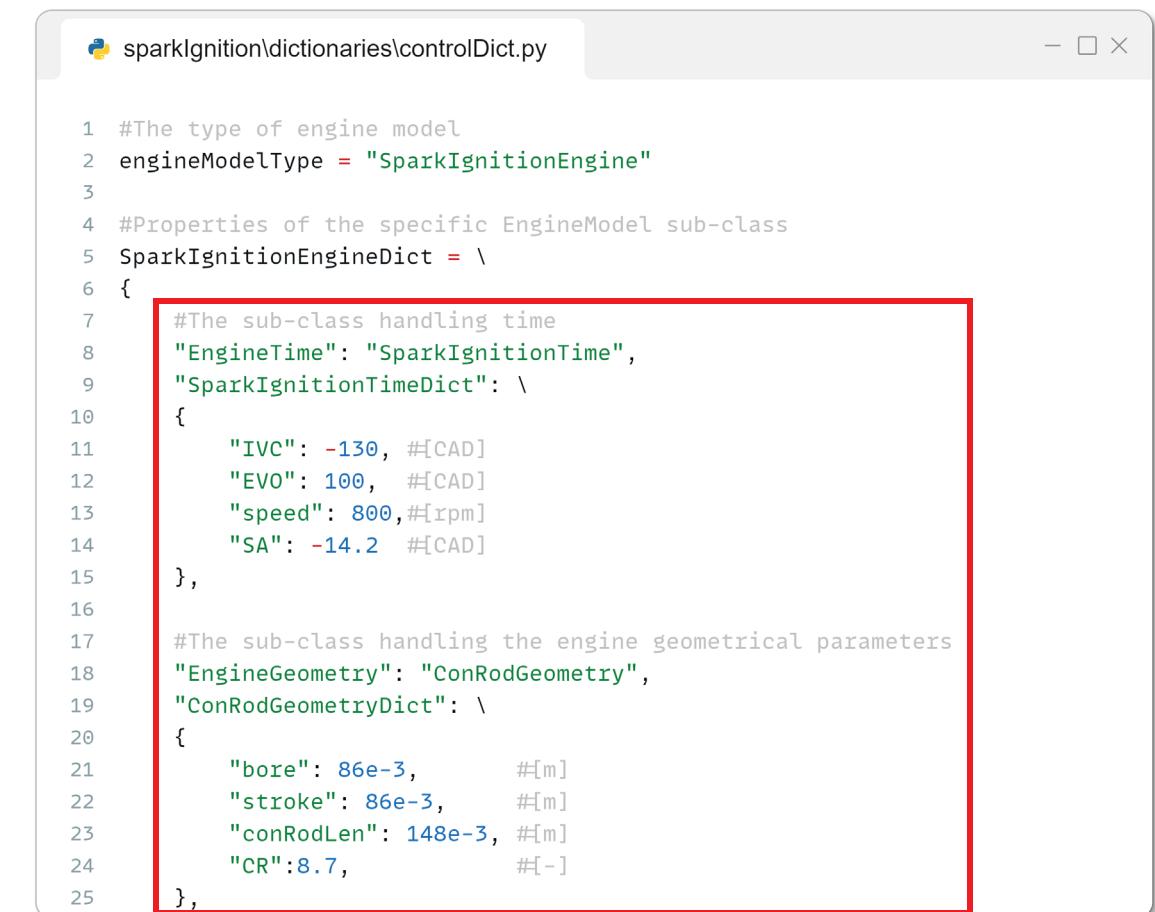
```

1 #The type of engine model
2 engineModelType = "SparkIgnitionEngine"
3
4 #Properties of the specific EngineModel sub-class
5 SparkIgnitionEngineDict = \
6 {
7     #The sub-class handling time
8     "EngineTime": "SparkIgnitionTime",
9     "SparkIgnitionTimeDict": \
10    {
11        "IVC": -130, #[CAD]
12        "EVO": 100, #[CAD]
13        "speed": 800,#[rpm]
14        "SA": -14.2 #[CAD]
15    },
16
17     #The sub-class handling the engine geometrical parameters
18     "EngineGeometry": "ConRodGeometry",
19     "ConRodGeometryDict": \
20    {
21        "bore": 86e-3,      #[m]
22        "stroke": 86e-3,    #[m]
23        "conRodLen": 148e-3, #[m]
24        "CR": 8.7,          #[-]
25    },

```

# Setup files – controlDict.py

- This control dictionary contains the specific engine model to use and the construction dictionary of the corresponding class.
- The **construction dictionary** must contain:
  - TypeName and construction dictionary (in the form <TypeName>Dict) for **EngineTime** and **EngineGeometry** sub-models



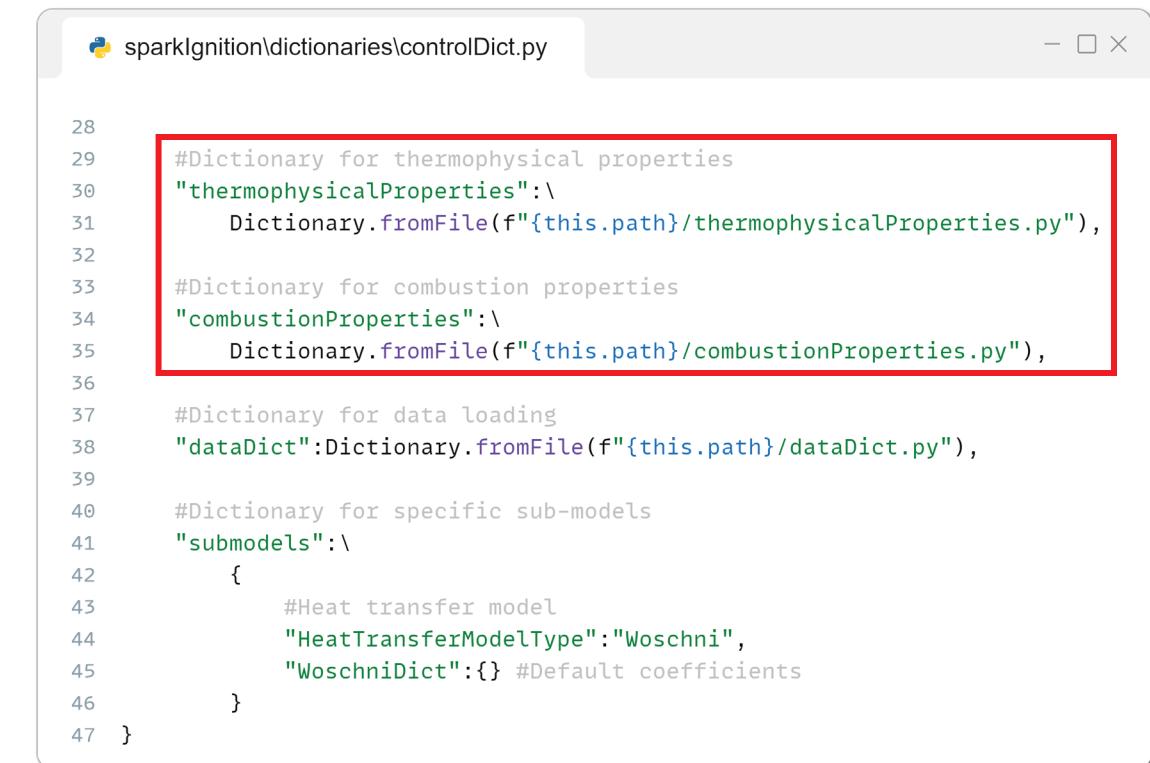
```

1 #The type of engine model
2 engineModelType = "SparkIgnitionEngine"
3
4 #Properties of the specific EngineModel sub-class
5 SparkIgnitionEngineDict = \
6 {
7     #The sub-class handling time
8     "EngineTime": "SparkIgnitionTime",
9     "SparkIgnitionTimeDict": \
10    {
11        "IVC": -130, #[CAD]
12        "EVO": 100, #[CAD]
13        "speed": 800,#[rpm]
14        "SA": -14.2 #[CAD]
15    },
16
17     #The sub-class handling the engine geometrical parameters
18     "EngineGeometry": "ConRodGeometry",
19     "ConRodGeometryDict": \
20    {
21        "bore": 86e-3,      #[m]
22        "stroke": 86e-3,   #[m]
23        "conRodLen": 148e-3, #[m]
24        "CR": 8.7,          #[-]
25    },

```

# Setup files – controlDict.py

- This control dictionary contains the specific engine model to use and the construction dictionary of the corresponding class.
- The **construction dictionary** must contain:
  - TypeName and construction dictionary (in the form <TypeName>Dict) for **EngineTime** and **EngineGeometry** sub-models
  - Two dictionaries containing the **thermophysical** and **combustion properties of mixtures**

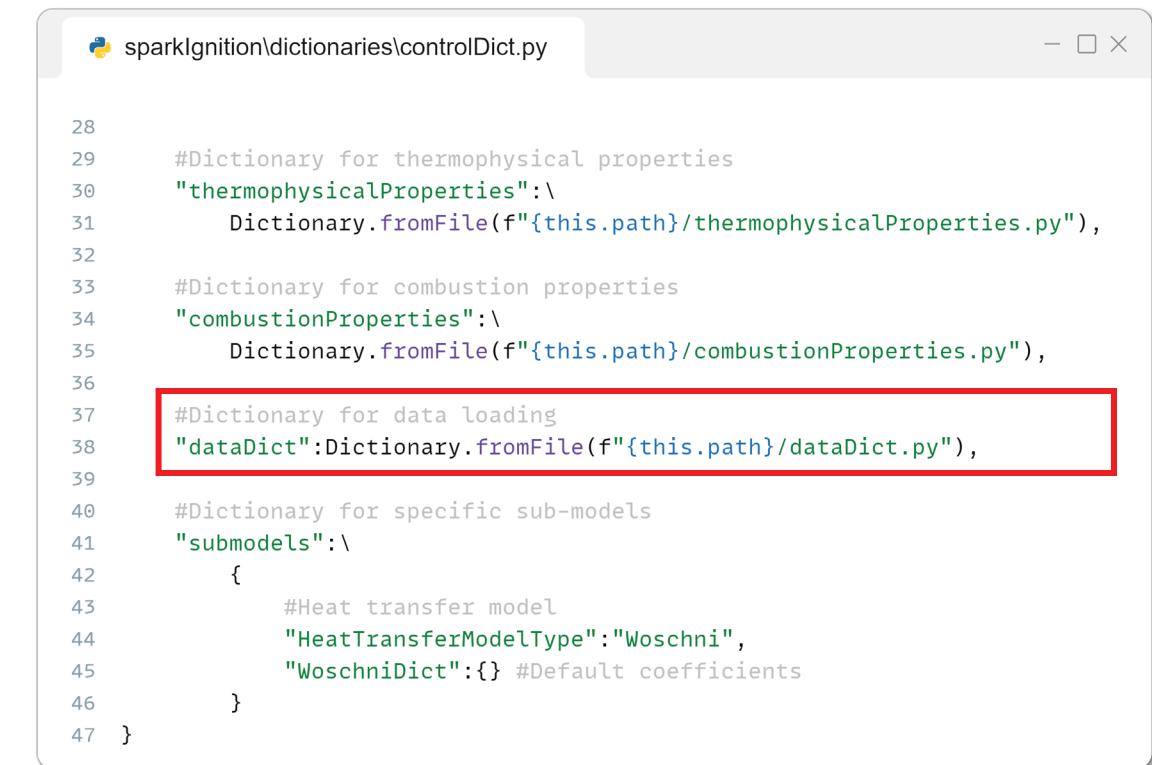


```

28
29 #Dictionary for thermophysical properties
30 "thermophysicalProperties":\
31     Dictionary.fromFile(f"{this.path}/thermophysicalProperties.py"),
32
33 #Dictionary for combustion properties
34 "combustionProperties":\
35     Dictionary.fromFile(f"{this.path}/combustionProperties.py"),
36
37 #Dictionary for data loading
38 "dataDict":Dictionary.fromFile(f"{this.path}/dataDict.py"),
39
40 #Dictionary for specific sub-models
41 "submodels":\
42 {
43     #Heat transfer model
44     "HeatTransferModelType":"Woschni",
45     "WoschniDict":{} #Default coefficients
46 }
47 }
```

# Setup files – controlDict.py

- This control dictionary contains the specific engine model to use and the construction dictionary of the corresponding class.
- The **construction dictionary** must contain:
  - TypeName and construction dictionary (in the form <TypeName>Dict) for **EngineTime** and **EngineGeometry** sub-models
  - Two dictionaries containing the thermophysical and combustion properties of mixtures
  - A dictionary containing the information for **loading input data** and **pre-processing** (filtering, scaling, etc.)

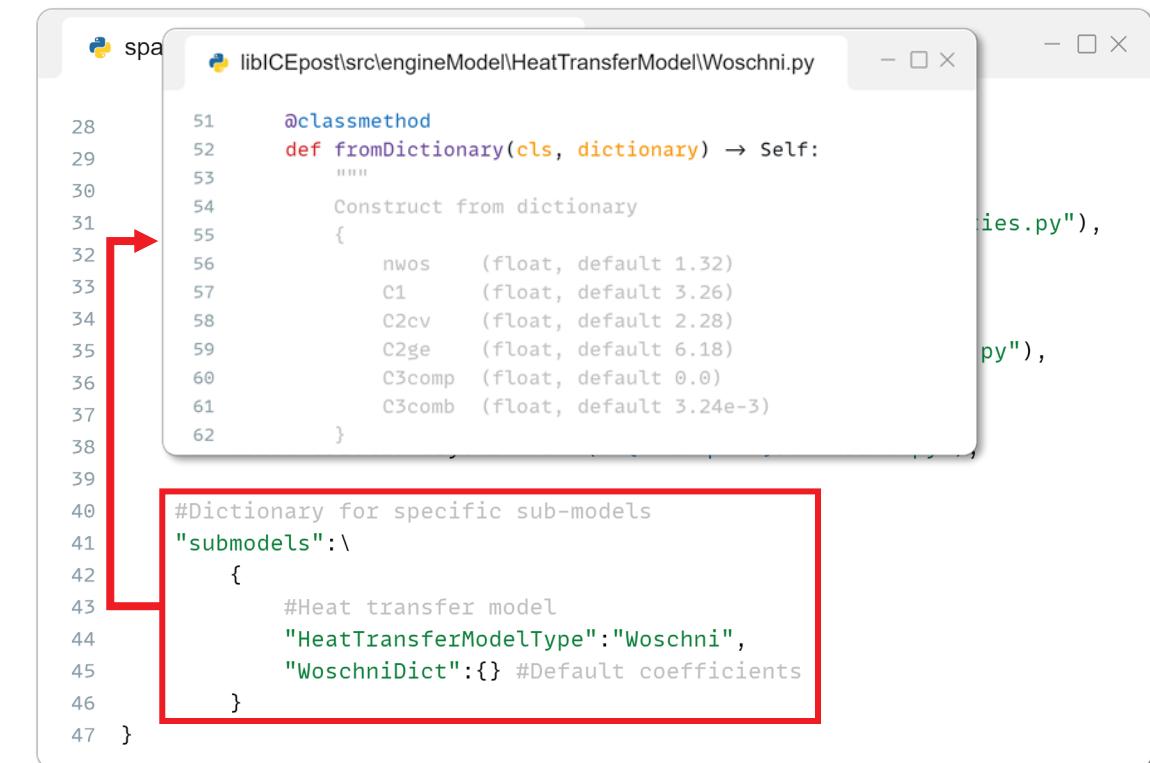


```

28
29     #Dictionary for thermophysical properties
30     "thermophysicalProperties":\
31         Dictionary.fromFile(f"{this.path}/thermophysicalProperties.py"),
32
33     #Dictionary for combustion properties
34     "combustionProperties":\
35         Dictionary.fromFile(f"{this.path}/combustionProperties.py"),
36
37     #Dictionary for data loading
38     "dataDict":Dictionary.fromFile(f"{this.path}/dataDict.py"),
39
40     #Dictionary for specific sub-models
41     "submodels":\
42     {
43         #Heat transfer model
44         "HeatTransferModelType":"Woschni",
45         "WoschniDict":{} #Default coefficients
46     }
47 }
```

# Setup files – controlDict.py

- This control dictionary contains the specific engine model to use and the construction dictionary of the corresponding class.
- The **construction dictionary** must contain:
  - TypeName and construction dictionary (in the form <TypeName>Dict) for **EngineTime** and **EngineGeometry** sub-models
  - Two dictionaries containing the thermophysical and combustion properties of mixtures
  - A dictionary containing the information for loading input data and pre-processing (filtering, scaling, etc.)
  - A dictionary containing the specific information of other sub-models (e.g. heat transfer)



```

 51     @classmethod
 52     def fromDictionary(cls, dictionary) → Self:
 53         """
 54             Construct from dictionary
 55         {
 56             nwos    (float, default 1.32)
 57             C1     (float, default 3.26)
 58             C2cv   (float, default 2.28)
 59             C2ge   (float, default 6.18)
 60             C3comp (float, default 0.0)
 61             C3comb (float, default 3.24e-3)
 62         }
 63
 64
 65     #Dictionary for specific sub-models
 66     "submodels":\"
 67     {
 68         #Heat transfer model
 69         "HeatTransferModelType": "Woschni",
 70         "WoschniDict": {} #Default coefficients
 71     }
 72 }
```

- **thermophysicalProperties:**

- **thermoType** – how to construct the **thermodynamic model of mixtures**, namely the equation of state and the thermal properties ( $c_p$ ,  $c_v$ ,  $h_f$ , etc.)

```
sparkIgnitionExample/controlDict.py
```

```
27 #Dictionary for thermophysical properties
28 "thermophysicalProperties":\
29     Dictionary.fromFile(f"{this.path}/thermophysicalProperties.py"),
30
31 #Dictionary for combustion properties
32 "combustionProperties":\
33     Dictionary.fromFile(f"{this.path}/combustionProperties.py"),
34
35 #Dictionary for data loading
36 "dataDict":Dictionary.fromFile(f"{this.path}/dataDict.py")
37
```

```
sparkIgnitionExample/thermophysicalProperties.py
```

```
1 #Classes to use for construction of the thermodynamic type
2 thermoType = \
3     {
4         #The equation of state to use
5         "EquationOfState":"PerfectGas", #Perfect gas assumption
6
7         #The thermodynamic properties for the mixture
8         "Thermo":"janaf7", # 7-coefficients NASA (janaf) polynomials
9     }
10
11 #Dictionaries for class-specific entries
12 PerfectGasDict = {}
13 janaf7Dict = {}
```

- **thermophysicalProperties:**

- thermoType – how to construct the thermodynamic model of mixtures, namely the equation of state and the thermal properties ( $c_p$ ,  $c_v$ ,  $h_f$ , etc.)
- Specific information required by the models

```
sparkIgnitionExample/controlDict.py
```

```
27 #Dictionary for thermophysical properties
28 "thermophysicalProperties":\
29     Dictionary.fromFile(f"{this.path}/thermophysicalProperties.py"),
30
31 #Dictionary for combustion properties
32 "combustionProperties":\
33     Dictionary.fromFile(f"{this.path}/combustionProperties.py"),
34
35 #Dictionary for data loading
36 "dataDict":Dictionary.fromFile(f"{this.path}/dataDict.py")
37
```

```
sparkIgnitionExample/thermophysicalProperties.py
```

```
1 #Classes to use for construction of the thermodynamic type
2 thermoType = \
3     {
4         #The equation of state to use
5         "EquationOfState":"PerfectGas", #Perfect gas assumption
6
7         #The thermodynamic properties for the mixture
8         "Thermo":"janaf7", # 7-coefficients NASA (janaf) polynomials
9     }
10
11 #Dictionaries for class-specific entries
12 PerfectGasDict = {}
13 janaf7Dict = {}
```

- **thermophysicalProperties:**

- **thermoType** – how to construct the **thermodynamic model of mixtures**, namely the equation of state and the thermal properties ( $c_p$ ,  $c_v$ ,  $h_f$ , etc.)
- Specific information required by the models

- **combustionProperties:**

```
sparkIgnitionExample/controlDict.py
27     #Dictionary for thermophysical properties
28     "thermophysicalProperties":\
29         Dictionary.fromFile(f"{this.path}/thermophysicalProperties.py"),
30
31     #Dictionary for combustion properties
32     "combustionProperties":\
33         Dictionary.fromFile(f"{this.path}/combustionProperties.py"),
34
35     #Dictionary for data loading
36     "dataDict":Dictionary.fromFile(f"{this.path}/dataDict.py")
37

sparkIgnitionExample/combustionProperties.py
1  #Loading mixtures and molecules from database
2  from libICEpost.Database.chemistry.specie.Mixtures import Mixtures, Mixture
3  from libICEpost.Database.chemistry.specie.Molecules import Fuels
4  air = Mixtures.dryAir    #Standard air composition
5
6  #Dictionary for computation of initial mixture composition in each region
7  initialMixture = \
8  {
9      #In-cylinder region
10     "cylinder": \
11         {
12             #Information for the premixed fuel
13             "premixedFuel": \
14                 {
15                     #The mixture of the fuel (pure iso-octane)
```

#Dictionary for combustion properties  
"combustionProperties":\nDictionary.fromFile(f"{this.path}/combustionProperties.py"),

#Loading mixtures and molecules from database  
from libICEpost.Database.chemistry.specie.Mixtures import Mixtures, Mixture  
from libICEpost.Database.chemistry.specie.Molecules import Fuels

**Loading properties from database**

- **thermophysicalProperties:**

- **thermoType** – how to construct the **thermodynamic model of mixtures**, namely the equation of state and the thermal properties ( $c_p$ ,  $c_v$ ,  $h_f$ , etc.)
- Specific information required by the models

- **combustionProperties:**

- **air** – The specific Mixture to use for air (oxidizer)

```
sparkIgnitionExample/controlDict.py
```

```
27 #Dictionary for thermophysical properties
28 "thermophysicalProperties":\
29     Dictionary.fromFile(f"{this.path}/thermophysicalProperties.py"),
30
31 #Dictionary for combustion properties
32 "combustionProperties":\
33     Dictionary.fromFile(f"{this.path}/combustionProperties.py"),
34
35 #Dictionary for data loading
36 "dataDict":Dictionary.fromFile(f"{this.path}/dataDict.py")
37
```

```
sparkIgnitionExample/combustionProperties.py
```

Using the entry  
from database

```
1 #Loading mixtures and molecules
2 from libICEpost.Database.chemistry.Species import Mixtures, Mixture
3 from libICEpost.Database.chemistry.Species.Molecules import Fuels
4 air = Mixtures.dryAir #Standard air composition
5
6 #Dictionary for computation of initial mixture composition in each region
7 initialMixture = \
8 {
9     #In-cylinder region
10    "cylinder": \
11        {
12            #Information for the premixed fuel
13            "premixedFuel": \
14                {
15                    #The mixture of the fuel (pure iso-octane)
```

- **thermophysicalProperties:**

- **thermoType** – how to construct the **thermodynamic model of mixtures**, namely the equation of state and the thermal properties ( $c_p$ ,  $c_v$ ,  $h_f$ , etc.)
- Specific information required by the models

- **combustionProperties:**

- **air** – The specific Mixture to use for air (oxidizer)
- **initialMixture** – A dictionary for **initialization of the mixture** inside each thermodynamic region

```
sparkIgnitionExample/controlDict.py
```

```
27 #Dictionary for thermophysical properties
28 "thermophysicalProperties":\
29     Dictionary.fromFile(f"{this.path}/thermophysicalProperties.py"),
30
31 #Dictionary for combustion properties
32 "combustionProperties":\
33     Dictionary.fromFile(f"{this.path}/combustionProperties.py"),
34
35 #Dictionary for data loading
36 "dataDict":Dictionary.fromFile(f"{this.path}/dataDict.py")
37
```

```
sparkIgnitionExample/combustionProperties.py
```

```
6 #Dictionary for computation of initial mixture composition in each region
7 initialMixture = \
8 {
9     #In-cylinder region
10    "cylinder": {
11        #Information for the premixed fuel
12        "premixedFuel": {
13            #The mixture of the fuel (pure iso-octane)
14            "mixture": Mixture([Fuels.IC8H18], [1.0]),
15
16            #The equivalence ratio of the mixture
17            "phi":1.0,
18        },
19    },
20}
```

Using the entry  
from database

- **thermophysicalProperties:**

- **thermoType** – how to construct the **thermodynamic model of mixtures**, namely the equation of state and the thermal properties ( $c_p$ ,  $c_v$ ,  $h_f$ , etc.)
- Specific information required by the models

- **combustionProperties:**

- **air** – The specific Mixture to use for air (oxidizer)
- **initialMixture** – A dictionary for initialization of the mixture inside each thermodynamic region
- The model to compute the exhaust gas composition

```
sparkIgnitionExample/controlDict.py
```

```
27     #Dictionary for thermophysical properties
28     "thermophysicalProperties":\
29         Dictionary.fromFile(f"{this.path}/thermophysicalProperties.py"),
30
31     #Dictionary for combustion properties
32     "combustionProperties":\
33         Dictionary.fromFile(f"{this.path}/combustionProperties.py"),
34
35     #Dictionary for data loading
36     "dataDict":Dictionary.fromFile(f"{this.path}/dataDict.py")
37
```

```
sparkIgnitionExample/combustionProperties.py
```

```
21
22 }
23
24 # The model to account for exhaust-gas recirculation
25 # on in-cylinder mixture composition
26 EgrModel = "StoichiometricMixtureEGR" #EGR composition computed at lambda = 1
27
28 StoichiometricMixtureEGRDict = \
29 {
30     "air":Mixtures.dryAir,
31     "fuel":Mixture([Fuels.IC8H18], [1.0]),
32     "egr":0.115,
33     "combustionEfficiency":0.9
34 }
35
36 # Giving separately the mass-fractions of externally
```

- **thermophysicalProperties:**

- **thermoType** – how to construct the **thermodynamic model of mixtures**, namely the equation of state and the thermal properties ( $c_p$ ,  $c_v$ ,  $h_f$ , etc.)
- Specific information required by the models

- **combustionProperties:**

- **air** – The specific Mixture to use for air (oxidizer)
- **initialMixture** – A dictionary for initialization of the mixture inside each thermodynamic region
- The model to compute the exhaust gas composition
- **The combustion model** to use to estimate the instantaneous mixture composition

```
sparkIgnitionExample/controlDict.py
```

```
27 #Dictionary for thermophysical properties
28 "thermophysicalProperties":\
29     Dictionary.fromFile(f"{this.path}/thermophysicalProperties.py"),
30
31 #Dictionary for combustion properties
32 "combustionProperties":\
33     Dictionary.fromFile(f"{this.path}/combustionProperties.py"),
34
35 #Dictionary for data loading
36 "dataDict":Dictionary.fromFile(f"{this.path}/dataDict.py")
37
```

```
sparkIgnitionExample/combustionProperties.py
```

```
38 ExternalInterEGRDict = \
39 {
40     "externalEGR":0.0,
41     "internalEGR":0.0,
42     "combustionEfficiency":0.9
43 }
44
45 #The combustion model used (spark-ignition)
46 CombustionModel = "PremixedCombustion" #Premixed combustion
47 PremixedCombustionDict = \
48 {
49     #The reaction model used to compute the mixture composition
50     "reactionModel":"Stoichiometry", # Stoichiometric-reaction
51                                         # combustion products
52                                         # [Reactants] → [Products]
53 }
```

# Setup files – dataDict

- **thermophysicalProperties:**
  - **thermoType** – how to construct the **thermodynamic model of mixtures**, namely the equation of state and the thermal properties ( $c_p$ ,  $c_v$ ,  $h_f$ , etc.)
  - Specific information required by the models
- **combustionProperties:**
  - **air** – The specific Mixture to use for air (oxidizer)
  - **initialMixture** – A dictionary for initialization of the mixture inside each thermodynamic region
  - The model to compute the exhaust gas composition
  - The **combustion model** to use to estimate the instantaneous mixture composition
- **dataDict:**
  - **dataPath** – The master path for input/output of data

```
sparkIgnitionExample/controlDict.py
```

```

27     #Dictionary for thermophysical properties
28     "thermophysicalProperties":\
29         Dictionary.fromFile(f"{this.path}/thermophysicalProperties.py"),
30
31     #Dictionary for combustion properties
32     "combustionProperties":\
33         Dictionary.fromFile(f"{this.path}/combustionProperties.py"),
34
35     #Dictionary for data loading
36     "dataDict":Dictionary.fromFile(f"{this.path}/dataDict.py")
37

```

```
sparkIgnitionExample/dataDict.py
```

```

1 #Global path where data are stored
2 dataPath = f"{this.path}/../data"
3
4 #The fields to load for each region
5 data = \
6 {
7     #Data for the cylinder region
8     "cylinder":\
9     {
10        #Pressure (mandatory)
11        "p":\
12        {
13            #The format of the data to load
14            "format":"file",      #Retreive from file
15

```

# Setup files – dataDict

- **thermophysicalProperties:**
  - **thermoType** – how to construct the **thermodynamic model of mixtures**, namely the equation of state and the thermal properties ( $c_p$ ,  $c_v$ ,  $h_f$ , etc.)
  - Specific information required by the models
- **combustionProperties:**
  - **air** – The specific Mixture to use for air (oxidizer)
  - **initialMixture** – A dictionary for initialization of the mixture inside each thermodynamic region
  - The model to compute the exhaust gas composition
  - The **combustion model** to use to estimate the instantaneous mixture composition
- **dataDict:**
  - **dataPath** – The master path for input/output of data
  - **data** – dictionary with the fields to be loaded for each thermodynamic region

```
sparkIgnitionExample/controlDict.py
```

```

27     #Dictionary for thermophysical properties
28     "thermophysicalProperties":\
29         Dictionary.fromFile(f"{this.path}/thermophysicalProperties.py"),
30
31     #Dictionary for combustion properties
32     "combustionProperties":\
33         Dictionary.fromFile(f"{this.path}/combustionProperties.py"),
34
35     #Dictionary for data loading
36     "dataDict":Dictionary.fromFile(f"{this.path}/dataDict.py")
37

```

```
sparkIgnitionExample/dataDict.py
```

```

1  #Global path where data are stored
2  dataPath = f"{this.path}../data"
3
4  #The fields to load for each region
5  data = \
6  {
7      #Data for the cylinder region
8      "cylinder":\
9      {
10          #Pressure (mandatory)
11          "p":\
12          {
13              #The format of the data to load
14              "format":"file",    #Retreive from file

```

# Setup files – dataDict

- **thermophysicalProperties:**
  - **thermoType** – how to construct the **thermodynamic model of mixtures**, namely the equation of state and the thermal properties ( $c_p$ ,  $c_v$ ,  $h_f$ , etc.)
  - Specific information required by the models
- **combustionProperties:**
  - **air** – The specific Mixture to use for air (oxidizer)
  - **initialMixture** – A dictionary for initialization of the mixture inside each thermodynamic region
  - The model to compute the exhaust gas composition
  - The **combustion model** to use to estimate the instantaneous mixture composition
- **dataDict:**
  - **dataPath** – The master path for input/output of data
  - **data** – dictionary with the fields to be loaded for each thermodynamic region

```
sparkIgnitionExample/controlDict.py
```

```

27     #Dictionary for thermophysical properties
28     "thermophysicalProperties":\
29         Dictionary.fromFile(f"{this.path}/thermophysicalProperties.py"),
30
31     #Dictionary for combustion properties
32     "combustionProperties":\
33         Dictionary.fromFile(f"{this.path}/combustionProperties.py"),
34
35     #Dictionary for data loading
36     "dataDict":Dictionary.fromFile(f"{this.path}/dataDict.py")
37

```

```
sparkIgnitionExample/dataDict.py
```

```

11     "p":\
12     {
13         #The format of the data to load
14         "format":"file",      #Retreive from file
15
16         #The parameters to use
17         "data":\
18             {
19                 #Name of the file
20                 "fileName":"p.Cyl",
21
22                     #Options to apply (scaling, shifting, etc.)
23                     "opts":{}
24             }
25     }

```

**Format for loading the variable.  
Multiple formats available (see example file for more)**

**File name (relative to dataPath)**

# Setup files – dataDict

- **thermophysicalProperties:**
  - **thermoType** – how to construct the **thermodynamic model of mixtures**, namely the equation of state and the thermal properties ( $c_p$ ,  $c_v$ ,  $h_f$ , etc.)
  - Specific information required by the models
- **combustionProperties:**
  - **air** – The specific Mixture to use for air (oxidizer)
  - **initialMixture** – A dictionary for initialization of the mixture inside each thermodynamic region
  - The model to compute the exhaust gas composition
  - The **combustion model** to use to estimate the instantaneous mixture composition
- **dataDict:**
  - **dataPath** – The master path for input/output of data
  - **data** – dictionary with the fields to be loaded for each thermodynamic region
  - **preProcessing** – manipulation of data before processing (resampling, denoising, etc.)

```

sparkIgnitionExample/controlDict.py
27 #Dictionary for thermophysical properties
28 "thermophysicalProperties":\
29     Dictionary.fromFile(f"{this.path}/thermophysicalProperties.py"),
30
31 #Dictionary for combustion properties
32 "combustionProperties":\
33     Dictionary.fromFile(f"{this.path}/combustionProperties.py"),
34
35 #Dictionary for data loading
36 "dataDict":Dictionary.fromFile(f"{this.path}/dataDict.py")
37

sparkIgnitionExample/dataDict.py
84
85 #Parameters for pre-processing of data
86 preprocessing = \
87 {
88     #Filter to apply on the data-set
89     "Filter":"Resample", # Resampling over uniform grid
90     "ResampleDict":\
91         {
92             "delta":1.0 #Delta-t [CA]
93         },
94
95     #User-defined filter through function/lambda such as:
96     #f(Iterable,Iterable) -> tuple[Iterable,Iterable]
97     "UserDefinedFilterDict":\
98         {
99             "function": lambda x,y : (x,y) #Dummy no-filter function

```

# Setup files – dataDict

- **thermophysicalProperties:**
  - **thermoType** – how to construct the **thermodynamic model of mixtures**, namely the equation of state and the thermal properties ( $c_p$ ,  $c_v$ ,  $h_f$ , etc.)
  - Specific information required by the models
- **combustionProperties:**
  - **air** – The specific Mixture to use for air (oxidizer)
  - **initialMixture** – A dictionary for initialization of the mixture inside each thermodynamic region
  - The model to compute the exhaust gas composition
  - The **combustion model** to use to estimate the instantaneous mixture composition
- **dataDict:**
  - **dataPath** – The master path for input/output of data
  - **data** – dictionary with the fields to be loaded for each thermodynamic region
  - **preProcessing** – manipulation of data before processing (resampling, denoising, etc.)

```
sparkIgnitionExample/controlDict.py
```

```

27     #Dictionary for thermophysical properties
28     "thermophysicalProperties":\
29         Dictionary.fromFile(f"{this.path}/thermophysicalProperties.py"),
30
31     #Dictionary for combustion properties
32     "combustionProperties":\
33         Dictionary.fromFile(f"{this.path}/combustionProperties.py"),
34
35     #Dictionary for data loading
36     "dataDict":Dictionary.fromFile(f"{this.path}/dataDict.py")
37

```

```
sparkIgnitionExample/dataDict.py
```

```

84
85 #Parameters for pre-processing of data
86 preprocessing = \
87 {
88     #Filter to apply on the data-set
89     "Filter":"Resample", # Resampling over uniform grid
90     "ResampleDict":\
91     {
92         "delta":1.0 #Delta-t [CA]
93     },
94
95     #User-defined filter through f
96     #f(Iterable,Iterable) -> tuple
97     "UserDefinedFilterDict":\
98     {
99         "function": lambda x,y

```

**Resampling of data over uniform grid. Optimal  $\Delta T$  of 1 CAD for accuracy and processing time**

# Setup files – dataDict

- **thermophysicalProperties:**
  - **thermoType** – how to construct the **thermodynamic model of mixtures**, namely the equation of state and the thermal properties ( $c_p$ ,  $c_v$ ,  $h_f$ , etc.)
  - Specific information required by the models
- **combustionProperties:**
  - **air** – The specific Mixture to use for air (oxidizer)
  - **initialMixture** – A dictionary for initialization of the mixture inside each thermodynamic region
  - The model to compute the exhaust gas composition
  - The **combustion model** to use to estimate the instantaneous mixture composition
- **dataDict:**
  - **dataPath** – The master path for input/output of data
  - **data** – dictionary with the fields to be loaded for each thermodynamic region
  - **preProcessing** – manipulation of data before processing (resampling, denoising, etc.)
  - **initialConditions** – initial state of each region

```

sparkIgnitionExample/controlDict.py
27     #Dictionary for thermophysical properties
28     "thermophysicalProperties":\
29         Dictionary.fromFile(f"{this.path}/thermophysicalProperties.py"),
30
31     #Dictionary for combustion properties
32     "combustionProperties":\
33         Dictionary.fromFile(f"{this.path}/combustionProperties.py"),
34
35     #Dictionary for data loading
36     "dataDict":Dictionary.fromFile(f"{this.path}/dataDict.py")
37

```

```

sparkIgnitionExample/dataDict.py
106             "delta":1.0 #Delta-t over which resampling after filtering
107
108         }
109     }
110
111     #Initial conditions each thermodynamic region
112     initialConditions = \
113     {
114         #In-cylinder region (at closed-valves)
115         "cylinder": \
116         {
117             "pressure":"p",           #Interpolation from field 'p'
118             "mass":"m_CFD",        #Interpolation from field 'm_CFD'
119             "volume":"@geometry.V", #From the method self.geometry.V(CA)
120             "xb":0.0,               #Value
121         }
122     }

```

**Generating the figure and axes: left axis for pressure trace and right for ROHR**

```
sparkIgnitionExample/main.py
47     'figure.figsize':(6,12),
48     # 'figure.dpi':10
49   })
50
51 #Create figure and combined axes (pressure lefr, ROHR right)
52 fig, axP = plt.subplots(1,1, figsize=(6,8))
53 axRohr = axP.twinx()
54
55 #Processed data are stored in the entry 'model.data' of type EngineData.
56 #This is a wrapper around a pandas.DataFrame data-structure, which is
57 #the entry EngineModel.data. Hence we use the method DataFrame.plot, which
58 #is a convenient interface for plotting of the data inside the DataFrame instance
59
60 #Pressure
61 model.data.data.plot(\n62     x="CA", y="pBar",\n63     c="k", ylabel="p [bar]", xlabel="[CAD]",\n64     ax=axP,\n65     legend=False)\n66
67 #Rate of heat release (ROHR)
68 model.data.data.plot(\n69     x="CA", y="ROHR",\n70     c="k", ls="--", ylabel="ROHR [J/CA]",\n71     ax=axRohr,\n72     legend=False)\n73
74 #Adjust axes and limits
75 plt.tight_layout()
76 axP.set_xlim((-30,30)) #x axis
77 axP.set_ylim((-10,40)) #pressure
78 axRohr.set_ylim((0,100)) #ROHR
79
80 #Show the figure
81 plt.show()
```

```
sparkIgnitionExample/main.py
47     'figure.figsize':(6,12),
48     # 'figure.dpi':10
49 }
50
51 #Create figure and combined axes (pressure lefr, ROHR right)
52 fig, axP = plt.subplots(1,1, figsize=(6,8))
53 axRohr = axP.twinx()
54
55 #Processed data are stored in the entry 'model.data' of type EngineData
56 #This is a wrapper around a pandas.DataFrame data-structure, which
57 #the entry EngineModel.data. Hence we use the method DataFrame.plot()
58 #is a convenient interface for plotting of the data inside the Data
59
60 #Pressure
61 model.data.data.plot(\n62     x="CA", y="pBar",\n63     c="k", ylabel="p [bar]", xlabel="[CAD]",\n64     ax=axP,\n65     legend=False)\n66\n67 #Rate of heat release (ROHR)
68 model.data.data.plot(\n69     x="CA", y="ROHR",\n70     c="k", ls="--", ylabel="ROHR [J/CA]",\n71     ax=axRohr,\n72     legend=False)\n73
74 #Adjust axes and limits
75 plt.tight_layout()
76 axP.set_xlim((-30,30)) #x axis
77 axP.set_ylim((-10,40)) #pressure
78 axRohr.set_ylim((0,100)) #ROHR
79
80 #Show the figure
81 plt.show()
```

Accessing the results at `model.data`, which is a [EngineData instance](#). Then accessing the `data` entry, which is of type [pandas.DataFrame](#), which has the convenient method `plot` for [visualizing the stored data](#)

```
sparkIgnitionExample/main.py
47     'figure.figsize':(6,12),
48     # 'figure.dpi':10
49 })
50
51 #Create figure and combined axes (pressure lefr, ROHR right)
52 fig, axP = plt.subplots(1,1, figsize=(6,8))
53 axRohr = axP.twinx()
54
55 #Processed data are stored in the entry 'model.data' of type EngineData.
56 #This is a wrapper around a pandas.DataFrame data-structure, which is
57 #the entry EngineModel.data. Hence we use the method DataFrame.plot, which
58 #is a convenient interface for plotting of the data inside the DataFrame instance
59
60 #Pressure
61 model.data.data.plot(
62     x="CA", y="pBar",                                     #x and y variables
63     c="k", ylabel="p [bar]", xlabel="[CAD]",             #Formatting
64     ax=axP,                                              #Axes
65     legend=False)                                         #No legend
66
67 #Rate of heat release (ROHR)
68 model.data.data.plot(
69     x="CA", y="ROHR",                                    #x and y variables
70     c="k", ls="--", ylabel="ROHR [J/CA]",              #Formatting
71     ax=axRohr,                                           #Axes
72     legend=False)                                         #No legend
73
74 #Adjust axes and limits
75 plt.tight_layout()
76 axP.set_xlim((-30,30)) #x axis
77 axP.set_ylim((-10,40)) #pressure
78 axRohr.set_ylim((0,100)) #ROHR
79
80 #Show the figure
81 plt.show()
```

Pressure trace on left axis

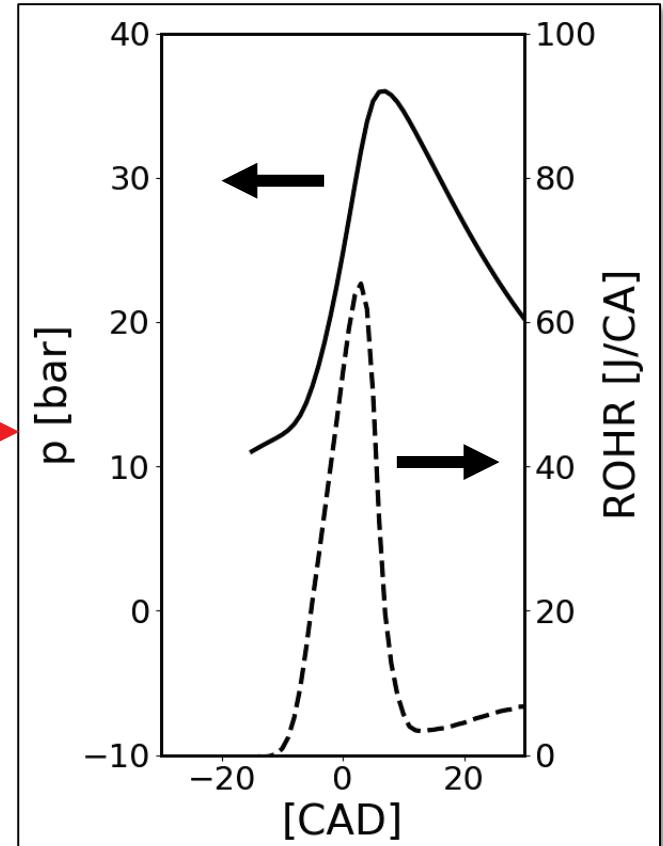
ROHR on right axis

```
sparkIgnitionExample/main.py
47     'figure.figsize':(6,12),
48     # 'figure.dpi':10
49 })
50
51 #Create figure and combined axes (pressure lefr, ROHR right)
52 fig, axP = plt.subplots(1,1, figsize=(6,8))
53 axRohr = axP.twinx()
54
55 #Processed data are stored in the entry 'model.data' of type EngineData.
56 #This is a wrapper around a pandas.DataFrame data-structure, which is
57 #the entry EngineModel.data. Hence we use the method DataFrame.plot, which
58 #is a convenient interface for plotting of the data inside the DataFrame instance
59
60 #Pressure
61 model.data.data.plot(
62     x="CA", y="pBar",                                     #x and y variables
63     c="k", ylabel="p [bar]", xlabel="[CAD]",             #Formatting
64     ax=axP,                                              #Axes
65     legend=False)                                         #No legend
66
67 #Rate of heat release (ROHR)
68 model.data.data.plot(
69     x="CA", y="ROHR",                                    #x and y variables
70     c="k", ls="--", ylabel="ROHR [J/CA]",              #Formatting
71     ax=axRohr,                                           #Axes
72     legend=False)                                         #No legend
73
74 #Adjust axes and limits
75 plt.tight_layout()
76 axP.set_xlim((-30,30)) #x axis
77 axP.set_ylim((-10,40)) #pressure
78 axRohr.set_ylim((0,100)) #ROHR
79
80 #Show the figure
81 plt.show()
```

Adjusting the axes and limits

```
sparkIgnitionExample/main.py
47     'figure.figsize':(6,12),
48     # 'figure.dpi':10
49 }
50
51 #Create figure and combined axes (pressure lefr, ROHR right)
52 fig, axP = plt.subplots(1,1, figsize=(6,8))
53 axRohr = axP.twinx()
54
55 #Processed data are stored in the entry 'model.data' of type EngineData.
56 #This is a wrapper around a pandas.DataFrame data-structure, which is
57 #the entry EngineModel.data. Hence we use the method DataFrame.plot, which
58 #is a convenient interface for plotting of the data inside the DataFrame instance
59
60 #Pressure
61 model.data.data.plot(
62     x="CA", y="pBar",
63     c="k", ylabel="p [bar]", xlabel="[CAD]",
64     ax=axP,
65     legend=False) #x and y variables
#Formatting
#Axes
#No legend
66
67 #Rate of heat release (ROHR)
68 model.data.data.plot(
69     x="CA", y="ROHR",
70     c="k", ls="--", ylabel="ROHR [J/CA]",
71     ax=axRohr,
72     legend=False) #x and y variables
#Formatting
#Axes
#No legend
73
74 #Adjust axes and limits
75 plt.tight_layout()
76 axP.set_xlim((-30,30)) #x axis
77 axP.set_ylim((-10,40)) #pressure
78 axRohr.set_ylim((0,100)) #ROHR
79
80 #Show the figure
81 plt.show()
```

Showing the plot



- **Next steps:**

- Extension of the classes to handle different engine configurations (Diesel engines, Wankel, etc...)
- Implementation of existing workflows into **executable scripts**
- Development of a **GUI** to simplify user experience

## Federico Ramognino

Department of Energy, Politecnico di Milano

Via Lambruschini, 4

20156 Milano, Italy

[federico.ramognino@polimi.it](mailto:federico.ramognino@polimi.it)



**PyPI:** <https://pypi.org/project/libICEpost/>



**GitHub:** <https://github.com/RamogninoF/LibICE-post>