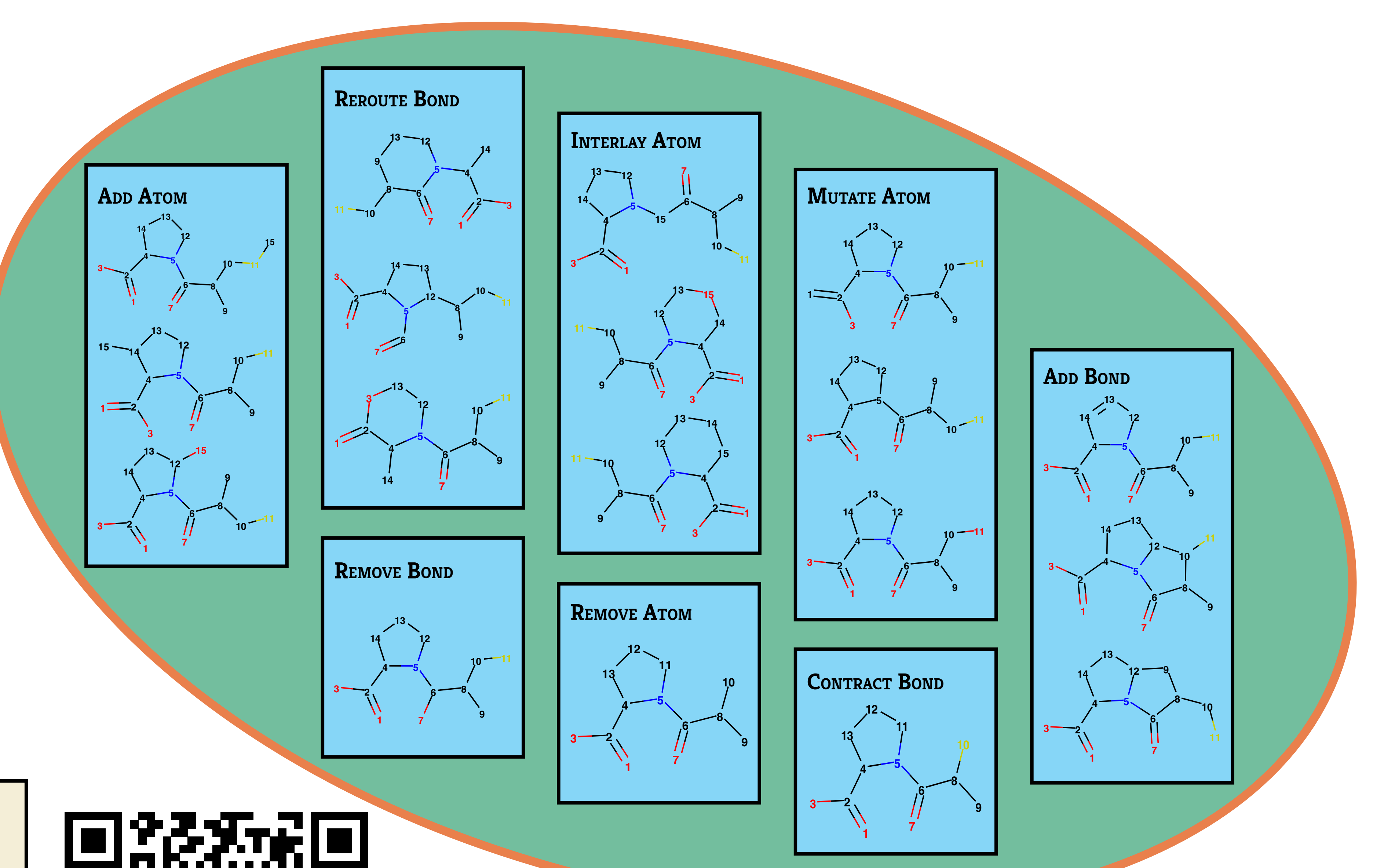
Molpher-lib: Programming Interface for Chemical Space Exploration

Šícho M.[1]*, Svozil D.*[1,2]

[1] *CZ-OPENSCREEN: National Infrastructure for Chemical Biology, Department of Informatics and Chemistry, Faculty of Chemical Technology, University of Chemistry and Technology Prague, Prague, Czech Republic.*

[2] *CZ-OPEN-SCREEN: National Infrastructure for Chemical Biology, Institute of Molecular Genetics, AS CR v.v.i., Prague, Czech Republic.*



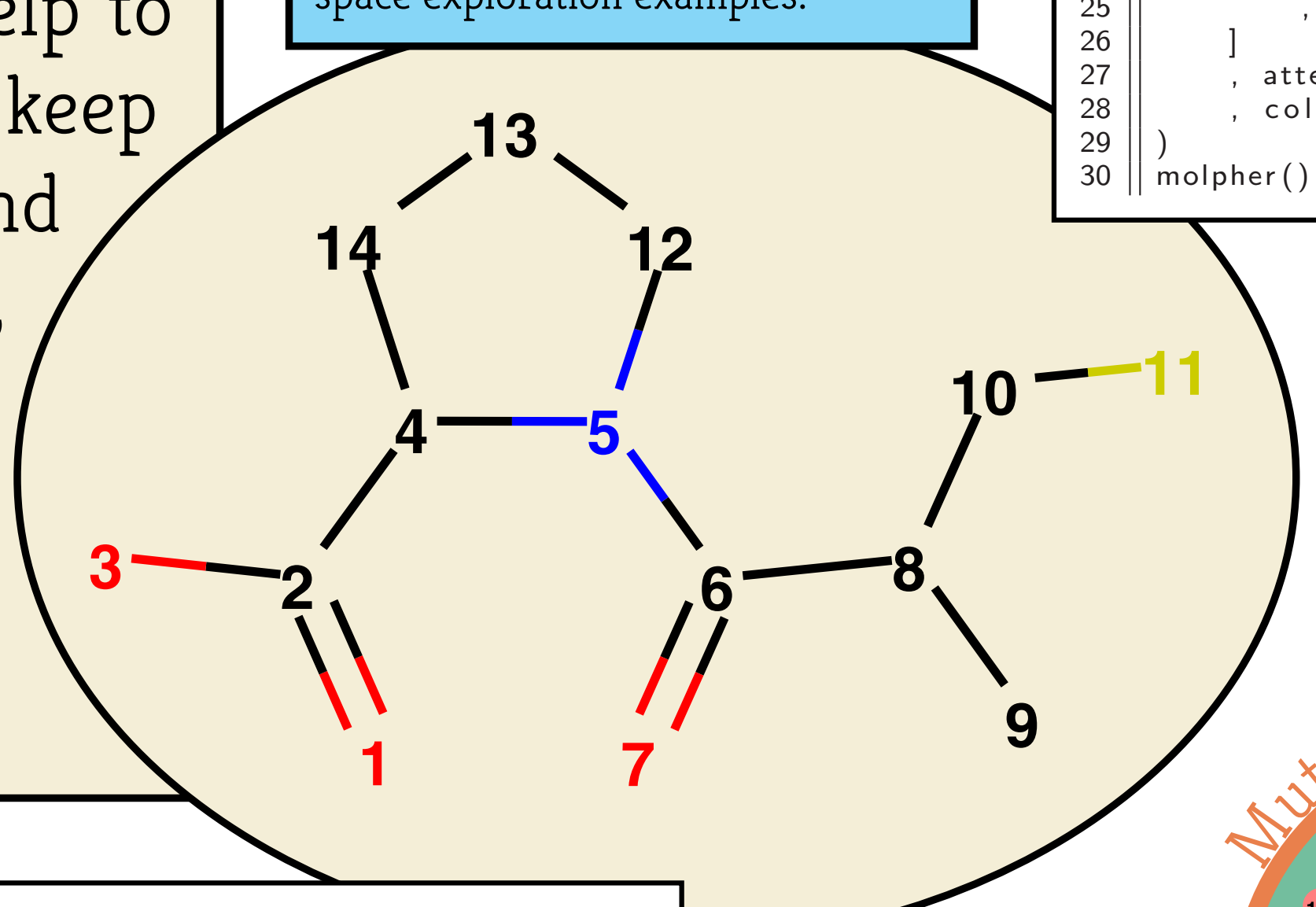
In silico Medicinal Chemistry with Molpher-lib

In rational drug design, discovery of new drugs would be impossible without the effort of medicinal chemists who often have to syntetize and test even hundreds of different analogs of a single lead compound to obtain a promising drug candidate. Such effort is time consuming and costly and it is often not clear what structural changes will actually lead to an improvment in the ADME properties of interest until a compound is prepared and tested.

Here, we present features of Molpher-lib, a chemical space exploration and structure generation tool, which (among other things) can now be used to simulate and automate certain medicinal chemistry tasks. The key features that help to facilitate this are (1) atom locking interface, which provides an option to keep certain atoms fixed and apply structural changes to the unlocked ones, and (2) an interface for implementation of customized morphing operators, which gives users the freedom to explore chemical space in a way most relevant to their chemical problem. These and other Molpher-lib features are demonstrated on an example structure of captopril, which can be seen on the right-hand side of this box.

Molpher-lib Website

This is captopril. It was the first ACE inhibitor in the treatment of hypertension. Its structure will be the starting point in our chemical space exploration examples.



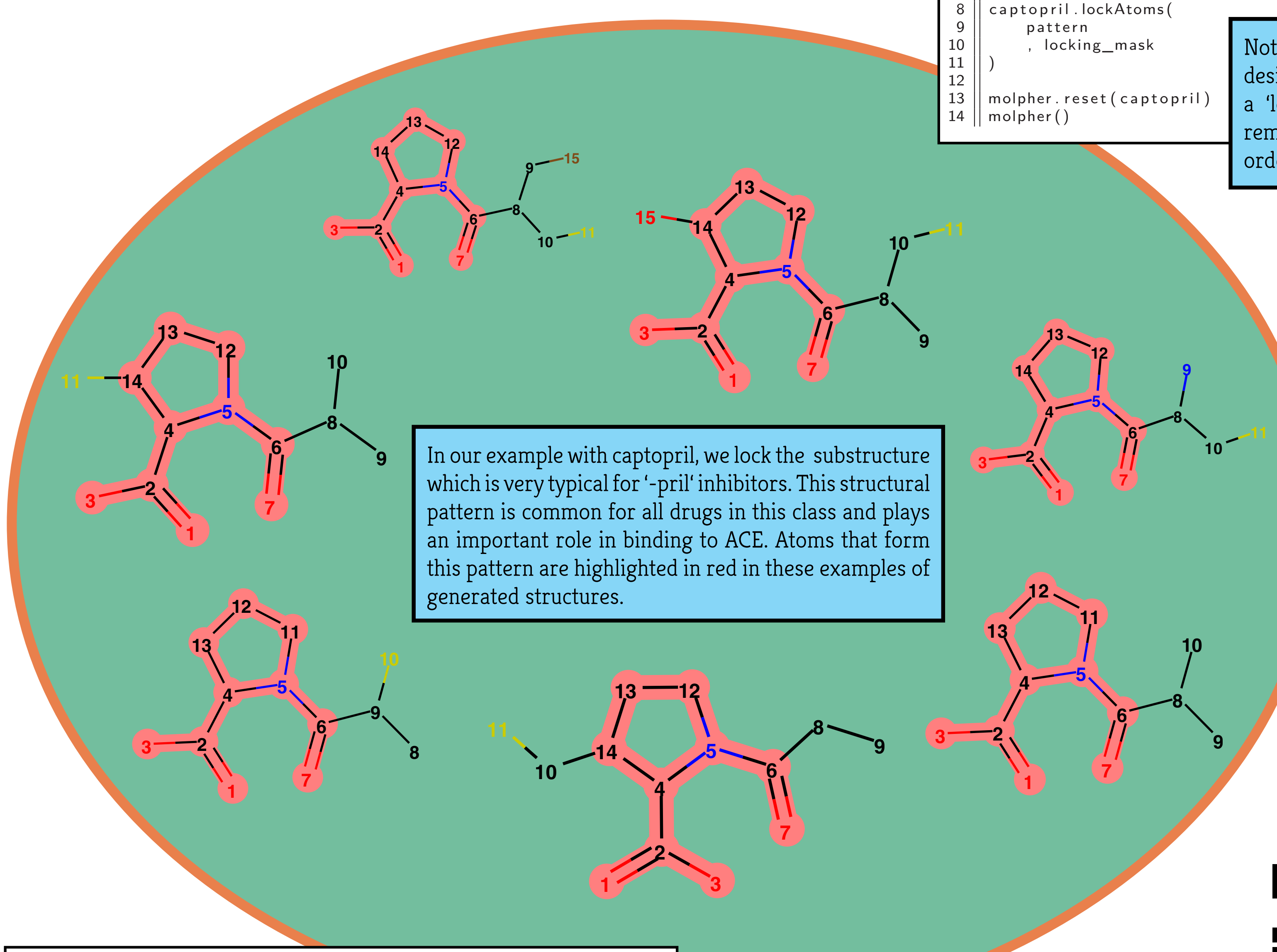
```python
import molpher
from molpher.core import MolpherMol # molecule representation
from molpher.core.morphing import Molpher # 'morphs' structures
from molpher.core.morphing.operators import * # built-in operators

generated_mols = dict() # stores generated molecules
def collect_unique(morph, oper):
    """
    simple function to collect generated morphs
    """
    generated_mols[morph.smiles] = morph


captopril = MolpherMol("captopril.sdf") # SMILES also possible
molpher = Molpher(
    captopril
    , operators = [
        AddAtom()
        , RemoveAtom()
        , MutateAtom()
        , InterlayAtom()
        , AddBond()
        , RemoveBond()
        , ContractBond()
        , RerouteBond()
    ]
    , attempts = 1000 # generate at most 1,000 structures
    , collectors = [collect_unique]
)
molpher() # start generating
```

Operators Docs

In Molpher-lib, the **Molpher** class handles transformations of the starting molecule to different structures by randomly selecting an 'operator' and applying it to the input structure, captopril in this case. Collector functions can then be used to examine the results and perform actions. In this example, we simply collect unique structures using canonical SMILES.

```python
from molpher.core import MolpherAtom

captopril = MolpherMol("captopril.sdf")

# atoms can be locked against certain modifications with a SMARTS pattern
pattern = "C1(C(=O)O)CCCN1C(=O)"
locking_mask = MolpherAtom.NO_REMOVAL | MolpherAtom.NO_MUTATION |
    MolpherAtom.KEEP_BONDS
captopril.lockAtoms(
    pattern
    , locking_mask
)

molpher.reset(captopril)
molpher()
```

Not all changes to the molecule are always desirable. Therefore, it is possible to specify a 'locking mask'. This mask will prevent removal of certain atoms, changes to bond order and other modifications.

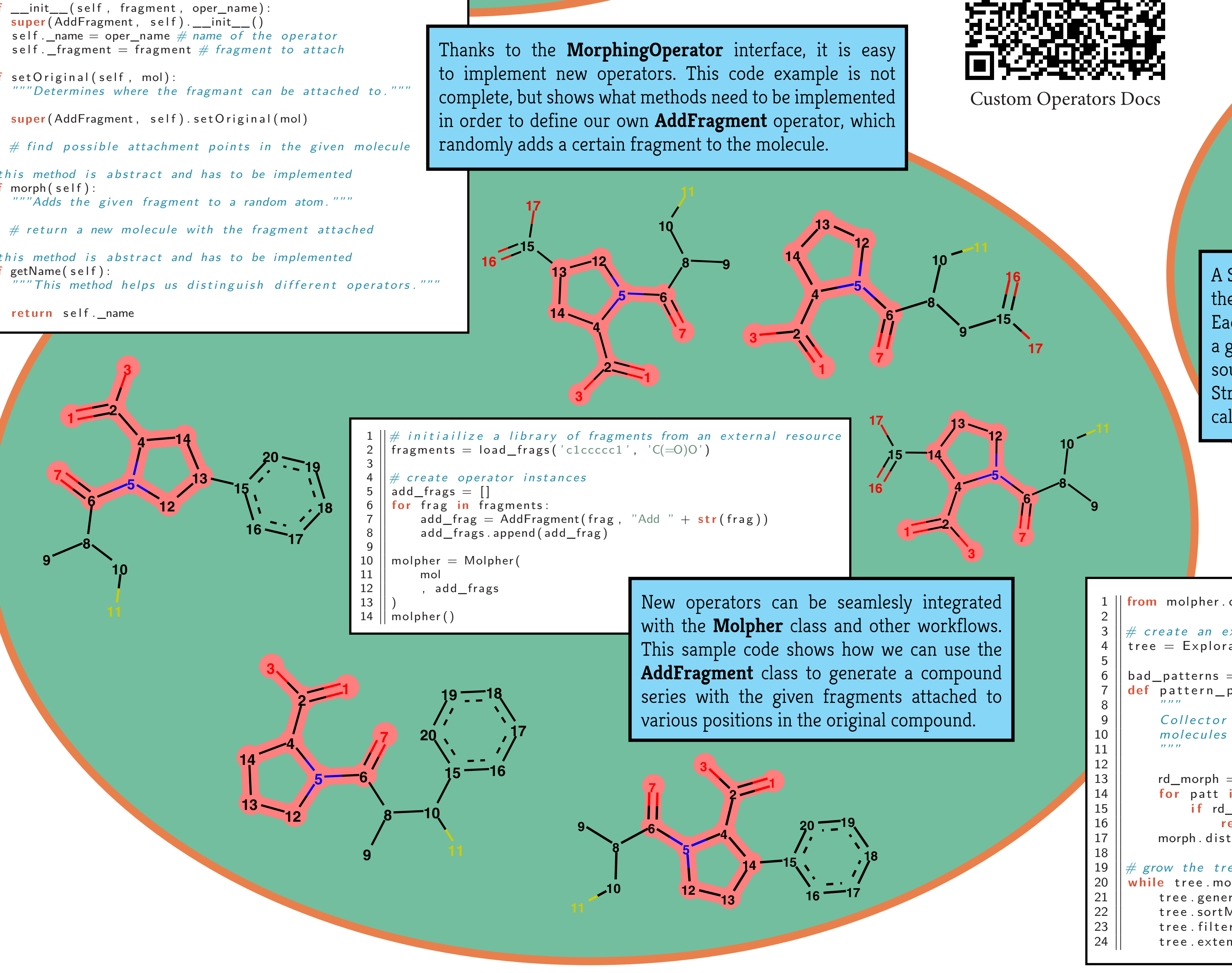


In our example with captopril, we lock the substructure which is very typical for '-pril' inhibitors. This structural pattern is common for all drugs in this class and plays an important role in binding to ACE. Atoms that form this pattern are highlighted in red in these examples of generated structures.

```python
class AddFragment(MorphingOperator):
    """Attaches a given fragment to an atom in the given molecule."""

    def __init__(self, fragment, oper_name):
        super(AddFragment, self).__init__()
        self._name = oper_name # name of the operator
        self._fragment = fragment # fragment to attach

    def setOriginal(self, mol):
        """Determines where the fragmant can be attached to."""

        super(AddFragment, self).setOriginal(mol)

        # find possible attachment points in the given molecule

    # this method is abstract and has to be implemented
    def morph(self):
        """Adds the given fragment to a random atom."""

        # return a new molecule with the fragment attached

    # this method is abstract and has to be implemented
    def getName(self):
        """This method helps us distinguish different operators."""

        return self._name
```

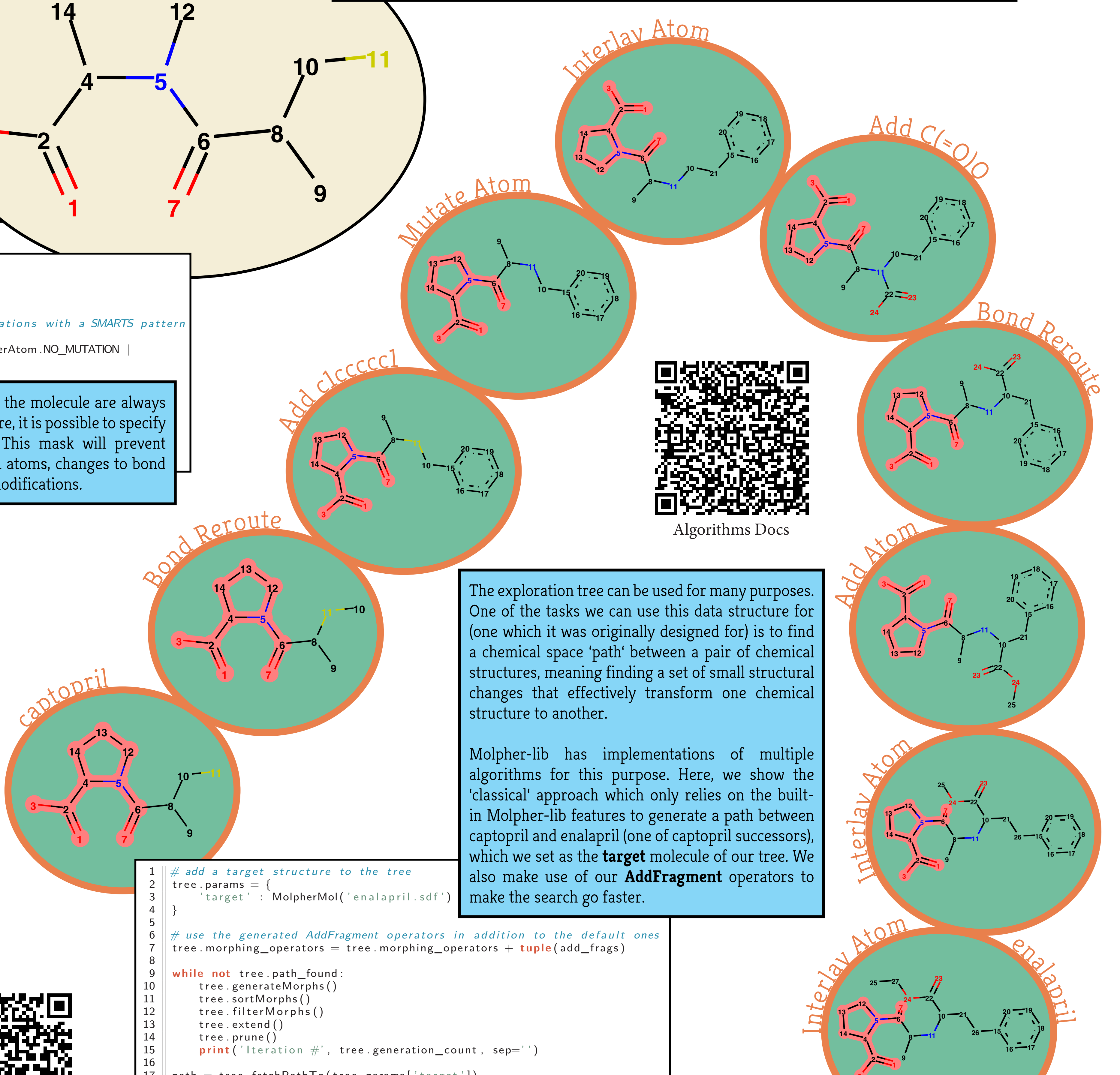
Thanks to the **MorphingOperator** interface, it is easy to implement new operators. This code example is not complete, but shows what methods need to be implemented in order to define our own **AddFragment** operator, which randomly adds a certain fragment to the molecule.

Custom Operators Docs



```python
# initiailize a library of fragments from an external resource
fragments = load_frags('c1ccccc1', 'C(=O)O')

# create operator instances
add_frags = []
for frag in fragments:
    add_frag = AddFragment(frag, "Add " + str(frag))
    add_frags.append(add_frag)

molpher = Molpher(
    mol
    , add_frags
)
molpher()
```

New operators can be seamlesly integrated with the **Molpher** class and other workflows. This sample code shows how we can use the **AddFragment** class to generate a compound series with the given fragments attached to various positions in the original compound.

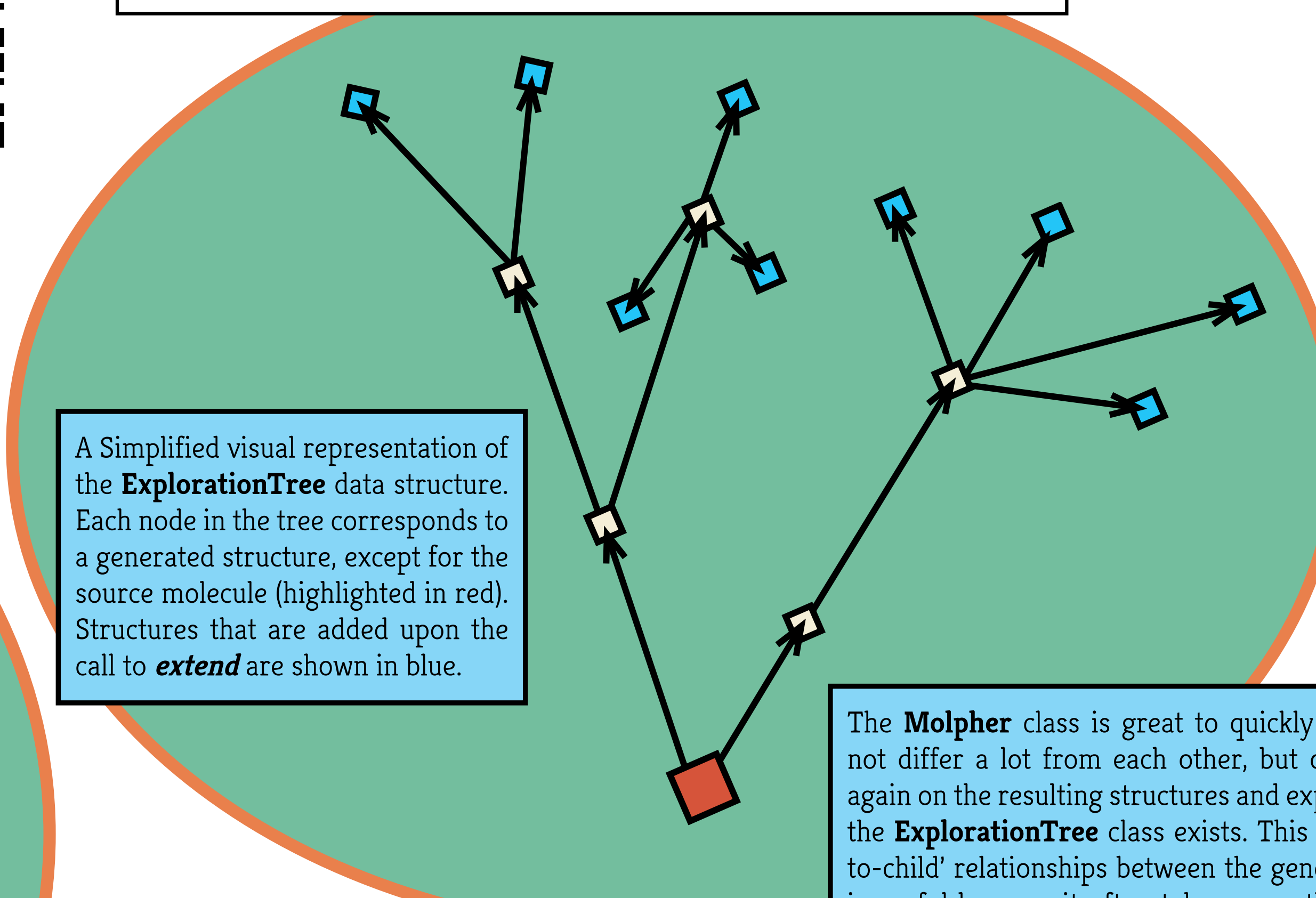


Algorithms Docs

The exploration tree can be used for many purposes. One of the tasks we can use this data structure for (one which it was originally designed for) is to find a chemical space 'path' between a pair of chemical structures, meaning finding a set of small structural changes that effectively transform one chemical structure to another.

Molpher-lib has implementations of multiple algorithms for this purpose. Here, we show the 'classical' approach which only relies on the built-in Molpher-lib features to generate a path between captopril and enalapril (one of captopril successors), which we set as the **target** molecule of our tree. We also make use of our **AddFragment** operators to make the search go faster.

```python
# add a target structure to the tree
tree.params = {
    'target' : MolpherMol('enalapril.sdf')
}

# use the generated AddFragment operators in addition to the default ones
tree.morphing_operators = tree.morphing_operators + tuple(add_frags)

while not tree.path_found:
    tree.generateMorphs()
    tree.sortMorphs()
    tree.filterMorphs()
    tree.extend()
    tree.prune()
    print('Iteration #', tree.generation_count, sep='')

path = tree.fetchPathTo(tree.params['target'])
```



A Simplified visual representation of the **ExplorationTree** data structure. Each node in the tree corresponds to a generated structure, except for the source molecule (highlighted in red). Structures that are added upon the call to ***extend*** are shown in blue.

Tree Docs

```python
from molpher.core import ExplorationTree

# create an exploration tree from the source molecule
tree = ExplorationTree.create(source=captopril)

bad_patterns = load_some_patterns()
def pattern_prioritize(morph, oper):
    """
    Collector that works with RDKit to prioritize
    molecules with 'better' structural patterns.
    """

    rd_morph = morph.asRDMol() # change to RDKit
    for patt in bad_patterns:
        if rd_morph.HasSubstructMatch(patt):
            return # this compound has no advantage -> do not prioritize
    morph.dist_to_target /= 2 # 'good' structures get 'closer'

# grow the tree until it has 10 thousand structures
while tree.mol_count < 10^4:
    tree.generateMorphs(collectors=[pattern_prioritize]) # generates new structures
    tree.sortMorphs() # sorts structures according to the 'dist_to_target' attribute
    tree.filterMorphs() # applies filters (selects structures from the top of the list)
    tree.extend() # extends the tree (connects selected structures to the tree)
```

The **Molpher** class is great to quickly generate a series of compounds that do not differ a lot from each other, but often we would like to use the operators again on the resulting structures and explore further in chemical space. That's why the **ExplorationTree** class exists. This data structure keeps track of the 'parent-to-child' relationships between the generated structures and their originals. This is useful because it often takes more than one structural modification to obtain a compound with suitable properties.

There are various built-in features that can be used to grow and manipulate the tree and also evaluate the generated structures in terms of chemical viability and the desired properties. In our example, we use four basic **ExplorationTree** methods to facilitate this: (1) ***generateMorphs*** to generate new candidates, (2) ***sortMorphs*** to create a sorted list of candidates in terms of their distances from the theoretical compounds we are looking for (represented by the **dist_to_target** attribute), (3) ***filter*** to remove unwanted candidates and (4) ***extend*** to connect selected molecules to the tree to modify them further. The ***pattern_prioritize*** collector acts as a simple objective function whose job is to make sure that 'interesting' compounds stay on top of the list when candidates are sorted.