# PyDesc tutorial

Tymoteusz Oleniecki

## 1 PyDesc - general information

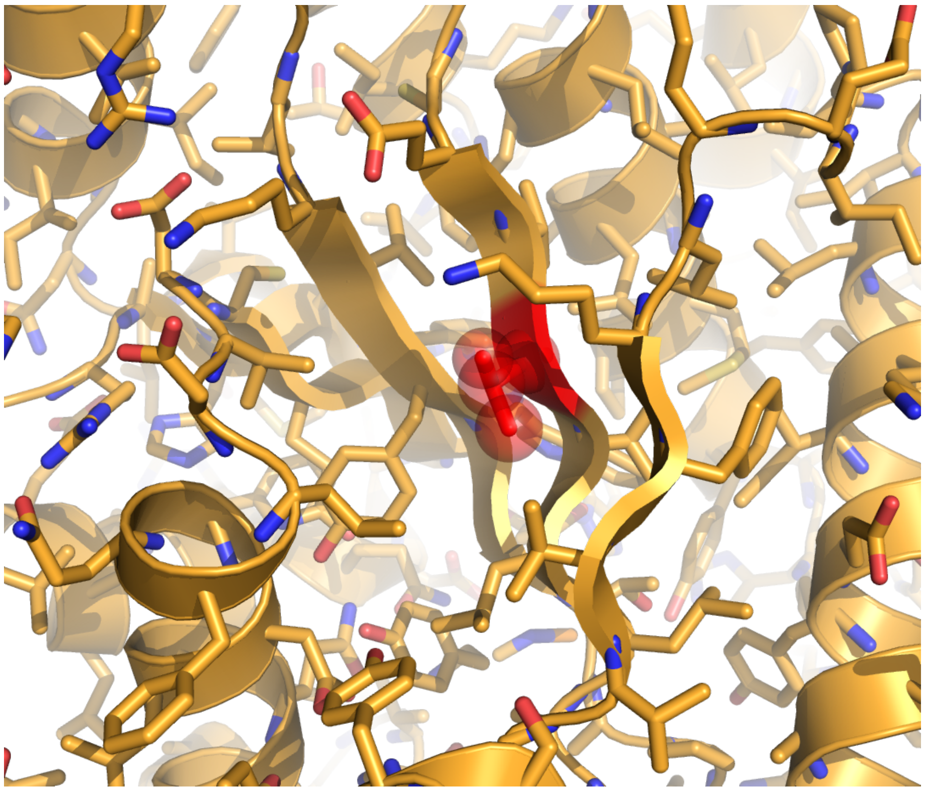
PyDesc is a Python 2.7.x library that provides classes which adapts Python environment to deal with biopolymer structures. It is designed for easy analysis of structures of both, proteins and nucleic acids, mostly using descriptors of local structure (DLS) approach (see chapter ??). Starting from very basic functionality as access to common databases of structures such as PDB, SCOP and CATH, and calculations of different geometrical features of structures or their parts, through combinations of contact criteria and calculation of contact maps, to the structural alignment of multiple structures in descriptor approach or analysis of MD trajectories – everything makes PyDesc useful for bioinformatic tools developers and for fast data analysis for scientists. Basic features of library are written in Python and available to developer for expansion or modification. More advanced and computationally demanding parts are written in C and CUDA C for performance boost. PyDesc requires BioPython for `pdb` parsing and numpy for most calculations. Additional features are enabled with DSSP, blastp, blastn and prody installed.

## 2 Basic features

PyDesc is able to load `pdb` files from common databases over Internet (PDB (including BioUnits), CATH, SCOP) or their local copies as well as single files from local drive. To deal with mmCIF files it loads bundle of `pdb` files in format provided by PDB. Loaded structures are stored in hierarchical objects, in which chains are stored in structures, mers are stored in chains and mers are simply containers for atoms and pseudoatoms useful in further analysis. It is also possible to create substructures of loaded structures such as ranges of sequential neighbours (called segments), sets of any mers or DLS (see chapter ??) for descriptor approach. Library provides friendly way to work with multiple structures. In case of homologous structures one can easily apply selections from one structure to another. As selection in PyDesc implements set theory approach different operations as addition or intersection of selections are easy to conduct. Dealing with non-homologous structures on the other hand is enabled thanks to modules providing alignment objects, which works on multiple alignments as well and allows to save and load files in common formats such as `FASTA`, `CSV` or `XML`. PyDesc provides calculation of contact maps and set of contact criteria to use for protein-protein and nucleic acid-nucleic acid interactions. It is also possible to combine provided contact criteria and to add user-defined criteria. That enables creation of DLS, which are substructures dependent on contact maps. Module for structural analysis, mostly written in C, provides structure comparison, (multiple) structure alignment and sequential/structural pattern fitting. Structure comparison allows for calculation of RMSD and rotation matrix using Kabsch method. It allows users to visualize superpositions (of both, pairs and multiple structures) using provided PyMOL plug-in. Calculation of structural alignment, by default, uses DLS approach and requires calculation of contact maps. Each of steps of structure analysis can be applied to frames of MD trajectories or states from NMR experiments.

## 3 Descriptors of Local Structure – DLS

Descriptors of Local Structure (DLS) are sets of mers – residues or nucleotides – representing local physicochemical environment of given mer. Such substructures are useful in structure comparison and computation of alignments, but can be useful in other structural analysis. In approaches using DLS structures are split into smaller overlapping fragments containing central mer (residue or nucleotide) from analyzed biopolymer and all other mers that stay in contact with it. That implies the size and shape of DLS depends on used criteria of contact between mers. The better those criteria describe real interactions between chemical compounds, the more useful obtained DLSs are as approximation of local environment. During calculation of DLS there are two main types of contacts: short and long range. Short range contacts are assumed between mer and their sequential neighbours, so preceding and following mers are included into DLS. Long range contacts depend on applied contact criteria. While defining criteria it is necessary to decide their type (quality, e.g. combination of distance and chemical properties etc.) and their thresholds (quantity, e.g. maximal distance, maximal difference in polarity etc.; see section XXX(ccrits)). Application of such criteria to all mers gives mers that stays in contact with central mer. All short range contacts of such mers are also included as they carry a information about physicochemical context of those mers.



## 4 Getting PyDesc

PyDesc is a library of classes written in Python 2.7. To install it, one needs Python 2.7 and dependent packages: BioPython, numpy and scipy. There are several way to download and install PyDesc:

- Use `git`

```
git clone https://github.com/pawelld/PyDesc.git
```

- Download `zip` package from [git repository].
- Use `pip` to install PyDesc from `egg` file included in repository.
- Use easy_install in downloaded directory:

```
easy_install setup.py
```

## 5 Getting started

PyDesc provides number of classes to deal with structural analysis. It can be used in development of Python scripts and programs, or as environment for fast analysis in Python interpreter. All the examples are ready to use in Python interpreter.

To check if PyDesc library is available simply import its components in Python:

```
import pydesc.sample
```

Loading structures in PyDesc requires class `StructureLoader` that is a component of `structure` module:

```
from pydesc.structure import StructureLoader
structure_loader = StructureLoader()
```

That class provides method `load_structure` to which user can pass reference to database entry or path to a file on local drive and get a Python list of structures from appropriate file:

```
from pydesc.structure import StructureLoader
structure_loader = StructureLoader()
structures1 = structure_loader.load_structure("2dlc")
structures2 = structure_loader.load_structure(code="2dlc")
structures3 = structure_loader.load_structure("pdb://2dlc")
structures4 = structure_loader.load_structure("2dlc",
    path="my/directory/2dlc.pdb")
```

In most cases file contains only one structure, but structures from NMR experiments or trajectories stored in `pdb` files are also able to be loaded using PyDesc. Structure 1 and 2 are downloaded automatically from first database to match the entry type format (`PDB` in that case). Structure 3 is loaded with forced usage of PDB. Structure 4 loads local file from `my/directory` and sets structure name to `2dlc`.

That is all the basic knowledge needed to load structures. Readers interested in quick start are welcome to skip to part **??**. Library features behind structure loading are described below.

During loader initialization User can choose what kind of database handler is to be used while loading structures. Database handler is an object responsible for connection to database. Available handlers are parts of `pydesc.dbhandler` package and are listed below:

- SCOPHandler – provides access to SCOP db.
- PDBHandler – provides access to `pdb` files from PDB.
- PDBBundleHandler – provides access to compatibility version of big `mmCIF` files available as bundle of `pdb` files with additional text file mapping chains.

- MMCIFHandler – provides access to `mmCIF` files from PDB.
- CATHHandler – provides access to CATH db.
- BioUnitHandler – provides access to BioUnit part of PDB.
- MetaHandler – meta handler containing all necessary handlers.

Objects of that classes can be passed to `StructureLoader` with following syntax:

```
from pydesc.structure import StructureLoader
from pydesc.dbhandler import PDBHandler
structure_loader = StructureLoader(PDBHandler())
structure_loader = StructureLoader(handler=PDBHandler())
# both lines are equivalents
```

By default `MetaHandler` is used.

When creating a dbhandler, bear in mind that they can work in three modes:

1. Always downloading files via Internet to local cache.
2. Always getting file from local copy of DB to local cache.
3. Always opening files from local cache.

Modes can be passed as a list to all handlers during their initialization. Subsequent modes will be executed one by one until first success or till the end of mode list, e.g.:

```
from pydesc.structure import StructureLoader
from pydesc.dbhandler import PDBHandler
structure_loader = StructureLoader(PDBHandler(mode=[1,2,3]))
structures = structure_loader.load_structure("2DLC")
```

which is equivalent to default setting, will 1) check local cache for `2dlc.pdb` file and load it if possible; otherwise it will 2) try to copy it from local DB copy to cache and load it again, but if that will end in failure, it will 3) download file from online DB to cache. Paths to local copy od DB and cache are stored in configuration manager (see chapter **??**).

Other argument passed to StructureLoader initializator is `parser`, which should be object with method `get_structure` returning instances of structures as defined in Biopython `Bio.PDB` module. By default `MetaParser` from `pydesc.dbhandler` module is used. That object contains `PDBParser` and `mmCIFParser` from Biopyton `Bio.PDB` module, instances of both classes can be used on their own, if only passed handler returns appropriate files. E.g. one can use `PDBHandler`, which provides only `pdb` files to parser, therefore parser could be `Bio.PDB.PDBParser`.

```
from pydesc.structure import StructureLoader
from pydesc.dbhandler import PDBHandler
from Bio.PDB import PDBParser
structure_loader = StructureLoader(handler=PDBHandler(),
    parser=PDBParser())
```

When using non-default parser make sure that it can handle type of files returned by used dbhandler.

## 6 Configuration

There is a bunch of global settings influencing PyDesc library behaviour. They are all stored in `pydesc.config` module in static class `ConfigManager` with `show_config` method that shows tree of configuration:

```
import pydesc.structure
from pydesc.config import ConfigManager

ConfigManager.show_config()
```

Output:

```
+ConfigManager
 +alignments
 ________-multiple_alignment_mode = strict
 +compdesc
 ________-ca_contact_distance = 6.0
 ________-force_order = 0
 ________-th_align_size_equiv = 3
...
```

All settings are stored in separate branches of names more or less corresponding with modules available in PyDesc. Branches containing appropriate configuration are created on module import. Settings important for certain function of library will be mentioned in appropriate chapters. Here we will present how to change and use settings using ezample of structure loaders mentioned in previous chapter.

```
import pydesc.structure
from pydesc.config import ConfigManager

print ConfigManager.dbhandler.cachedir
```

Setting `cachedir` in `dbhandler` branch of configuration manager is a directory in which cached structure files are stored. Lets assume that User conducts an experiment on set of structures. At some point he decides to test some other parameters in new directory, and wants to run modified scripts on the same set of structures. To avoid downloading and caching the same structures in new directory, User adds the following line:

```
ConfigManager.dbhandler.cachedir = '../old/directory/biodb/'
# equivalent:
ConfigManager.dbhandler.cachedir.set('../old/directory/biodb/')
```

## 7 Dealing with structures

Let us assume that one has already loaded a structure:

```
from pydesc.structure import StructureLoader
structure_loader = StructureLoader()
structures = structure_loader.load_structure("2dlc")
structure = structures[0]
```

As doing so in Python interpreter every time just to test some simple commands is arduous, for the sake of tuition PyDesc provides `get` function in `sample` module, which can be used instead.

```
from pydesc.sample import get
structure = get("2dlc")
```

That function returns only one structure from list of loaded structures. By default it is the first one, but User can pass index of structure to be returned as second argument. Note that lists in Python are indexed from 0.

## 7.1 Getting mers and substructures

PyDesc structures and substructures consist of mers (residue, nucleotide, ligands or ions), which consist of atoms. Each entity is represented by separate iterable object, so one way to get to building blocks of any level is to loop over their elements:

```
for mer in structure:
    print mer
    for atom in mer:
        print atom
```

Output:

```
<Residue: GLN no. 1, PDB: A6>
<Atom C   : 18.040001 6.891000 -2.685000>
<Atom CA  : 16.721001 7.525000 -3.144000>
<Atom CB  : 15.546000 6.793000 -2.485000>
<Atom CD  : 13.952000 4.840000 -2.513000>
<Atom CG  : 15.147000 5.500000 -3.170000>
<Atom N   : 16.635000 7.521000 -4.591000>
<Atom NE2: 14.089000 3.561000 -2.181000>
<Atom O   : 18.969000 6.780000 -3.474000>
<Atom OE1: 12.928000 5.490000 -2.312000>
<Pseudoatom : 16.721001 7.525000 -3.144000>
<Pseudoatom cbx: 14.779633 6.315571 -2.055182>
<Pseudoatom rc: 15.780779 6.100111 -2.950556>
```

Often we are not interested in whole structure, but in only one chain of it. Objects representing chains are created automatically and are stored in `chains` attribute of `Structure` instance. Class `Structure` also provides `get_chain` method, which takes chain name as `str`:

```
print structure.chains
print structure.get_chain('X')
```

Output:

```
[<Chain 2dlcY>, <Chain 2dlcX>]
<Chain 2dlcX>
```

Sample structure `2DLC` consists of chains "X" and "Y".

Chains in PyDesc are very similar to whole structures and any other part of structure we will deal with in that tutorial. All of them provide method for getting their mers using Python syntax, which behaves slightly different than its equivalent for python sequential types:

```
chainX = structure.get_chain('X')
print structure[0], structure[0].ind
print structure[1], structure[1].ind
print chainX[0], chainX[0].ind
print chainX[75], chainX[75].ind
```

Output:

```
<Nucleotide:   C no. 1, PDB: Y501> 1
<Nucleotide:   C no. 1, PDB: Y501> 1
<Residue: ASP no. 75, PDB: X8> 75
<Residue: ASP no. 75, PDB: X8> 75
```

Mers in PyDesc are indexed from 1. Index 0 always returns first mer of the (sub)structure. PyDesc mer id, stored in `ind` attribute of mer, is created once at the creation of structure the mer belongs to and identifies mer in that particular **structure**, thus picking chain from the middle of the structure leads to shifted indexing. Attempts to pick mer of index not occurring in (sub)structure leads to `IndexError`. This is very useful in some cases but can be confusing for new Users. In above example chain X starts with mer 75. If one wants to index elements of PyDesc structure as in Python, simply convert PyDesc structure to Python sequential type, e.g. list or tuple:

```
print tuple(chainX)[0], list(chainX)[1]
```

Output:

```
<Residue: ASP no. 75, PDB: X8> <Residue: PRO no. 76, PDB: X9>
```

Users are allowed to get parts or slices of structures:

```
seg = structure[75: 82]
for mer in seg: print mer
```

Note that mer 82 is a part of returned `Segment` instance! That is another difference between PyDesc and Python sequential types indexing. That means that expressions:

```
seg1 = structure[75: 75]
mer75 = structure[75]
```

return different type of objects. First one returns `Segment` instance of length 1, containing only 75th mer. Second one returns mer itself, residue in that case.

Structures can be easily joined using `+` operator:

```
mix = structure[75: 82] + structure[84:85] + structure[90:90]
```

This kind of substructures are instances of `UserStructure` objects and has all the methods already presented here for `Chain`, `Segment`, `Structure` classes.

Note that adding single mer to substructure requires adding 1-mer length segment (*structure*[90 : 90]). Attempt to add mer (*structure*[90]) will end up in `AttributeError`.

## 7.2 Getting atoms and pseudoatoms

Mers are sets of atoms and pseudoatoms (like geometrical center of side chain). Iteration over mer instance gives subsequent atoms and pseudoatoms:

```
for atom in structure[90]:
    print atom
```

Output:

```
<Atom C   : -2.385000 36.616001 173.134995>
<Atom CA  : -2.743000 37.533001 171.970001>
<Atom CB  : -3.075000 38.953999 172.513000>
<Atom CG1: -1.882000 39.539001 173.264999>
<Atom CG2: -3.498000 39.891998 171.386993>
<Atom N   : -3.867000 36.938000 171.257996>
<Atom O   : -3.216000 36.372002 174.016998>
<Pseudoatom : -2.813667 36.223667 171.880325>
<Pseudoatom cbx: -3.288228 39.866638 172.861740>
<Pseudoatom rc: -2.952286 37.977715 172.506424>
```

To get certain atom, one can use syntax similar to one used in Python dictionaries:

```
structure[90]['CA']
```

It works also for pseudoatoms, which usually are also an attribute of mer:

```
structure[90]['cbx']
structure[90].cbx
```

`cbx` is a pseudoatom defined for residues used by PyDesc default contact definition. It is vector $< C_\alpha, C_\beta >$ extended by 1 Å.

There are also alternative iterators for mer, available using mer methods:

- `__iter__` – default, iterating over all atoms and pseudoatoms.
- `iter_atoms` – iterating over atoms only.
- `iter_atomsbb` – iterating over backbone atoms only.
- `iter_atomsnbb` – iterating over non-backbone atoms only.

Distinguish of backbone and non-backbone atoms bases on setting `backbone_atoms` in branch `ConfigManager.monomer.*`, where asterisk is to be replaced with name of chainable mer, basically `residue` or `nucleotide`. Configuration stores any sequential type containing names of atoms as strings (with no spaces). E.g.:

```
from pydesc.config import monomer
from pydesc.config import ConfigManager

print ConfigManager.monomer.residue.backbone_atoms
```

Output:

```
('N', 'CA', 'C')
```

## 7.3 DLS, Contacts, Elements

There is also special kind of substructures containing parts of structure that are not sequential neighbours: DLSs (see chapter ??). That kind of structures depend on contact maps, which are matter of next chapter (??), here we will use default settings to calculate them.

To create DLS one needs to provide `Element` subclass instance (`ElementChainable`, `ElementOther`) of central mer and elements of all mers that stay in contact with central mer. Element chainable is special kind of `Segment` of odd number of mers with distinguished central mer. By default elements are segments of length 5 and depends on setting `pydesc.config.ConfigManager.element.element_chainable_length`. Elements of mers in contact with central mer should delivered as attributes of `Contact` instances, so second object needed to form a DLS is list of contacts. All classes, `Element` adn subclasses, `Contact` , as well as `ProteinDescriptor` or `NucleotideDescriptor` can be imported from `pydesc.structure` module.

Assuming that one knows the list of PyDesc indexes of mers that stays in contact with certain mer (list `cnt1` in example below), lets say 90 in `2DLC`, creation of descriptor goes as follow:

```
cnt1 = [84, 87, 93, 96, 280, 281, 282]

from pydesc.structure import ElementChainable
from pydesc.structure import Contact
from pydesc.structure import ProteinDescriptor

element90 = ElementChainable(structure[90])
descriptor90 = ProteinDescriptor(element90,
    [Contact(ElementChainable(structure[i]), element90) for i in
    cnt1])
```

Contacts could be read from contact map calculated with PyDesc module.

Similarly to this, one could create a nucleotide descriptor using `NucleotideDescriptor` class with appropriate element and contacts list. As that procedure is complicated and could be executed very often, each subclass of `AbstractDescriptor` provide static method of descriptor building basing on contact map calculated for structure from which mer comes from:

```
from pydesc.structure import StructureLoader
from pydesc.structure import Element
from pydesc.structure import ProteinDescriptor

structure = StructureLoader().load_structure('2dlc')[0]
structure.set_contact_map()

element90 = Element(structure[90])
descriptor90 = ProteinDescriptor.build(element90)
```

It is also possible to pass other contact map to that method as second argument. For structures containing both, protein and nucleic acids, it may be convenient to use method `build` provided by `AbstractDescriptor` class, which takes element and decides what type of descriptor to build:

```
from pydesc.structure import StructureLoader
from pydesc.structure import Element
from pydesc.structure import AbstractDescriptor

structure = StructureLoader().load_structure('2dlc')[0]
structure.set_contact_map()

element90 = Element.build(structure[90])  #residue
descriptor90 = AbstractDescriptor.build(element90)

element20 = Element.build(structure[25])  #nucleotide
descriptor20 = AbstractDescriptor.build(element20)

print descriptor90
print descriptor20
```

Output:

```
<ProteinDescriptor of 2dlc:X23>
<NucleotideDescriptor of 2dlc:Y526>
```

Often it is necessary to create all possible DLSs for given structure, which can be easily done with `create_descriptors` static method, which returns generator for subsequent DLSs:

```
from pydesc.structure import StructureLoader
from pydesc.structure import AstractDescriptor

structure = StructureLoader().load_structure('2dlc')[0]
structure.set_contact_map()
DLSgen = AbstractDescriptor.create_descriptors(structure)
DLSs = list(DLSgen)
```

For mers, for which DLS cannot be created `None` is returned. That allows for convenient usage of `zip` function on result:

```
DLSgen = AbstractDescriptor.create_descriptors(structure)
for mer, its_DLS in zip(structure, DLSgen):
    print mer, its_DLS
```

Output:

```
<Nucleotide:   C no. 1, PDB: Y501> None
<Nucleotide:   U no. 2, PDB: Y502> None
<Nucleotide:   C no. 3, PDB: Y503> <NucleotideDescriptor of
    2dlc:Y503>
<Nucleotide:   U no. 4, PDB: Y504> <NucleotideDescriptor of
    2dlc:Y504>
<Nucleotide:   C no. 5, PDB: Y505> <NucleotideDescriptor of
    2dlc:Y505>
<Nucleotide:   G no. 6, PDB: Y506> <NucleotideDescriptor of
    2dlc:Y506>
```

```
...
```

## 8 Selections

Selections are classes that allows users to apply set theory operations to structures and substructures. In other words – thanks to `selection` module it is possible to use structures as sets of mers, which they are. What is more – it allows to generalize operations on structures with consistent PDB numbering. Lets see how it works. There are two crystals of amicyjanins with different asymmetric units: `1sfd` and `2gb1`:

```
from pydesc.structure import StructureLoader

stc_1sfd = StructureLoader().load_structure('1sfd')[0]
stc_2gb1 = StructureLoader().load_structure('2gb1')[0]
```

Proteins are mutated, therefore they differ in positions 52 and 94. What is more – `1sfd` consists of two chains. Let us pick the whole loop and $\beta$-strand containing mutated residue 94 in `1sfd`:

```
from pydesc.selection import Range

sele = Range('A92', 'A96', distinguish_chains=False)
```

We can also get it using method of `Segment` class:

```
sele = stc1['A92':'A96'].select(distinguish_chains=False)
```

What is the difference between `Range` and `Segment` then? Selection object does not store any structural data, only information that selected mers are between IDs 92 and 96 (in chain A if argument `distinguish_chains` was set to True). It can be applied to any structure and used to get substructure or to create new independent structure basing on the old one:

```
loop1sfdA = sele.select(stc_1sfd.get_chain('A'))
loop1sfdB = sele.select(stc_1sfd.get_chain('B'))
loops1sfd = sele.select(stc_1sfd)
loop2gb1 = sele.select(stc_2gb1)
```

`Range` is probably the most useful selection, but most basic is `Set`. It takes list of residue IDs and can be useful in cases where we deal with structures consisting of mers with insertion codes. To convert range selection to set selection one needs to give a structure on which that conversion is to be made:

```
seleSet = sele.specify(stc_1sfd)
```

In range defined in `sele` in structure `stc_1sfd` there are no residues with insertion codes. But one may want to apply that selection to other structure. Applying a range selection will embrace all residues, while applying set selection obtained from range selection will pick only residues of IDs present in structure that was used as base to set creation.

All kinds of selections are listed below, with arguments demanded by initializator and respective class that returns that kind of selection upon call of its `select` method:

- Set, list of PDB IDs (`str`), `UserStructure`, `Monomer`
- Range, starting and ending PDB IDs (`str`), `Segment`
- ChainSelection, chain name (`str`), `Chain`
- MonomerName, PDB residue name (`str`), `Segment`
- MonomerType, `Monomer` subclass, –
- Everything, –, `Structure`
- Nothing, –, –

Basic operation known from set theory are provided:

```
from pydesc.selection import Everything
from pydesc.selection import MonomerType
from pydesc.selection import MonomerName
from pydesc.selection import ChainSelection

sele1 = MonomerName('TYR')        # selecting all tyrosines
sele2 = Everything() - sele1      # everything but tyrosines
sele3 = ChainSelection('A')       # chain A
sele4 = sele3 * sele2             # chain A except tyrosines
```

# 9 Contact maps

## 9.1 Basics, default criteria

Contact map contains information about contacts between mers in structure. To create one, one needs to know criteria of contact. First let us look at contact maps with default PyDesc criteria. PyDesc provides methods for checking values, getting contacts and saving all non-zero values in `CSV` file:

```
from pydesc.structure import StructureLoader

structure = StructureLoader().load_structure('2dlc')[0]
structure.set_contact_map()

print structure.contact_map
with open("2dlc_cmap.csv", "w") as fhobj:
    structure.contact_map.dump(fhobj)

print structure.contact_map.get_monomer_contacts(90)

print structure.contact_map.get_contact_value(90, 50)
print structure.contact_map.get_contact_value(90, 96)
print structure.contact_map.get_contact_value(90, 281)
```

Output:

```
[(90, 96, 1), (90, 281, 2), (90, 84, 2), (90, 87, 2), (90, 280,
    1), (90, 89, 2), (90, 282, 1), (90, 91, 2), (90, 92, 1), (90,
    93, 2)]
0
1
2
```

Data stored in contact map can be passed to any file-like object including streams and files. It calls `write` method on passed object and feeds it with string containing as manly lines, as many contacts there is in structure (times two, as they are assumed to be symmetrical), in each line placing two PDB ids and *value* that can be 1 or 2.

As contact criteria in PyDesc uses tree-value logic the latter mean reliable contact, and the former – possible contact. Zero denote no contact. Criterion usually consist of some quality requirements (what features of mer should be taken into account) and quality requirements (threshold, below which criterion is satisfied). By default PyDesc uses:

- for residues of indexes $i$ and $j$:

  $$distance(C_{\alpha,i}, C_{\alpha,j}) \leq 6.00 \pm 0.50\text{Å}$$

  OR

  $$distance(C_{\beta x,i}, C_{\beta x,j}) \leq 6.50 \pm 0.50\text{Å}$$

  $$\wedge$$

  $$distance(C_{\beta x,i}, C_{\beta x,j}) - distance(C_{\alpha,i}, C_{\alpha,j}) \leq 0.75 \pm 0.00\text{Å}$$

  where:
  $C_{\alpha,a}$ – coordinates of alpha carbon of $a$-th residue,
  $C_{\beta x,a}$ – coordinates of extended beta carbon of $a$-th residue.

- for nucleotides of indexes $i$ and $j$:

  $$distance(R_{c,i}, R_{c,j}) \leq 6.25 \pm 0.00\text{Å}$$

  OR

  $$distance(R_{c,i}, I) \leq 7.40 \pm 0.00\text{Å} \wedge distance(R_{c,j}, I) \leq 7.40 \pm 0.00\text{Å}$$

  where:
  $R_{c,a}$ – $a$-th nucleotide ring center,
  $I$ – ion coordinates.

Standard error notation was used here to indicate range in which contacts get value of 1.

### 9.2 Simple and combined pre-defined criteria

All pre-defined contact criteria and abstract classes are implemented in `contacts` module. For example first part of default criterion for residues measures distance of their alpha carbons and checks if it is under the threshold. It could be used alone for contact map calculation:

```
from pydesc.structure import StructureLoader
from pydesc.contacts import CaContact

structure = StructureLoader().load_structure('2dlc')[0]
structure.set_contact_map(CaContact())
print structure.contact_map.get_monomer_contacts(90)
```

Output:

```
[(90, 281, 2), (90, 280, 1), (90, 89, 2), (90, 282, 1), (90, 91,
    2), (90, 92, 1), (90, 93, 1)]
```

Note that with such criterion there is no contacts between any nucleotides (residues 1-74 for `2DLC`).

By default that criterion uses threshold equals to 6.00 and so called *undecidable range* of 0.50, which can be changed simply by passing them to criterion object initializator:

```
structure.set_contact_map(CaContact(8., .75))
print structure.contact_map.get_monomer_contacts(90)
```

Output:

```
[(90, 279, 1), (90, 281, 2), (90, 280, 2), (90, 84, 2), (90, 87,
    1), (90, 88, 2), (90, 89, 2), (90, 282, 2), (90, 91, 2), (90,
    92, 2), (90, 93, 2)]
```

If no arguments is passed to initializator, settings from configuration manager:

```
structure.set_contact_map(CaContact())
print structure.contact_map.get_monomer_contacts(90)
ConfigManager.contacts.ca_contact_distance = 8.0
ConfigManager.contacts.ca_contact_undecidable_range = 0.75
structure.set_contact_map(CaContact())
print structure.contact_map.get_monomer_contacts(90)
```

Output:

```
[(90, 281, 2), (90, 280, 1), (90, 89, 2), (90, 282, 1), (90, 91,
    2), (90, 92, 1), (90, 93, 1)]
[(90, 279, 1), (90, 281, 2), (90, 280, 2), (90, 84, 2), (90, 87,
    1), (90, 88, 2), (90, 89, 2), (90, 282, 2), (90, 91, 2), (90,
    92, 2), (90, 93, 2)]
```

Difference between calling initializator with arguments and changing default settings is in behaviour of already existing contact criterion:

```
ConfigManager.contacts.ca_contact_distance = 8.0
ConfigManager.contacts.ca_contact_undecidable_range = 0.75
ca_crit_SD = CaContact()
ca_crit_8_75 = CaContact(8., .75)

structure.set_contact_map(ca_crit_SD)
print structure.contact_map.get_monomer_contacts(90)
structure.set_contact_map(ca_crit_8_75)
print structure.contact_map.get_monomer_contacts(90)
```

Output:

```
[(90, 279, 1), (90, 281, 2), (90, 280, 2), (90, 84, 2), (90, 87,
    1), (90, 88, 2), (90, 89, 2), (90, 282, 2), (90, 91, 2), (90,
    92, 2), (90, 93, 2)]
[(90, 279, 1), (90, 281, 2), (90, 280, 2), (90, 84, 2), (90, 87,
    1), (90, 88, 2), (90, 89, 2), (90, 282, 2), (90, 91, 2), (90,
    92, 2), (90, 93, 2)]
```

Result is the same, but in case of any change in settings:

```
ConfigManager.contacts.ca_contact_distance = 6.0
ConfigManager.contacts.ca_contact_undecidable_range = 0.50

structure.set_contact_map(ca_crit_SD)
print structure.contact_map.get_monomer_contacts(90)
structure.set_contact_map(ca_crit_8_75)
print structure.contact_map.get_monomer_contacts(90)
```

Output:

```
[(90, 279, 1), (90, 281, 2), (90, 280, 2), (90, 84, 2), (90, 87,
    1), (90, 88, 2), (90, 89, 2), (90, 282, 2), (90, 91, 2), (90,
    92, 2), (90, 93, 2)]
[(90, 279, 1), (90, 281, 2), (90, 280, 2), (90, 84, 2), (90, 87,
    1), (90, 88, 2), (90, 89, 2), (90, 282, 2), (90, 91, 2), (90,
    92, 2), (90, 93, 2)]
```

the `ca_crit_SD` will change values on the fly.

Second part of default criterion is a combined criterion, first part of which is basically the same as previously described `CaContct`, just calculating distance between `cbx` with different threshold. For that purpose there is another class `CbxContact` in appropriate module. One could be interested in criterion a bit easier than default one: alternative of `ca` i `cbx` criteria. With class `ContactsAlternative` it is as simple as:

```
from pydesc.contacts import ContactsAlternative
from pydesc.contacts import CaContact
from pydesc.contacts import CbxContact

ccrit = ContactsAlternative(CaContact(), CbxContact())
```

One can pass as many contact criteria as needed. PyDesc delivers also `ContactsConjunction` and `ContactExclusiveDisjunction` classes, which can also be called with Python operators:

```
from pydesc.contacts import CaContact
from pydesc.contacts import CbxContact

alt = CaContact() | CbxContact()  #alternative
con = CaContact() & CbxContact()  #conjunction
exd = CaContact() ^ CbxContact()  #exclusive disjuntion
```

It is also possible to easily enerate negative criteria with `Not` class decorator, which for combined criteria reqires a bit tricky syntax:

```
from pydesc.contacts import ContactsAlternative
from pydesc.contacts import CaContact
from pydesc.contacts import CbxContact

n_Ca = Not(CaContact)()
n_alt = Not(ContactsAlternative)(CaContact(), CbxContact())
```

## 9.3 Defining own simple criteria

In most cases it is sufficient to use pre-defined classes of criteria. PyDesc provides following **abstract** (one should not create instance of those classes, only create classes that inherit from them) classes:

- `ContactCriterion` – generic criterion.
- `PointsDistanceCriterion` – criteria basing on distance between (pseudo)atoms of choice.
- `SetDistanceCriterion` – criteria based on set of distances between more than two pairs of atoms from different mers.
- `VectorDistanceCriterion` – criteria based on orientation of vectors distinguished in mers.
- `DihedralAngleCriterion` – criteria based on dihedral angels between planes spanned on mer atoms.

`PointsDistanceCriterion` is the class of most widely used type of geometrical criteria. Extending this class is the best way to get own criteria dependent on distance between (pseudo)atom of choice. For example, lets define criterion for side chain geometrical center, which is stored in `rc` attribute of any chainable mer (for sake of example, as that criterion is implemented in `RcContact` class):

```
from pydesc.contacts import PointsDistanceCriterion

class RcCrt(PointsDistanceCriterion):
    monomer_hallmark = 'rc'

crt = RcCrt(4., 1.)
```

It requires only static attribute `monomer_hallmark` set to string containing name of attribute to be used in distance calculation. Of course, that is more likely to be useful for types of mers defined by Users, with their own pseudoatoms.

If no arguments will be passed to initializator – class will automatically seek for settings in configuration manager in `pydesc.config.ConfigManager.contacts` branch. One important setting is `*attribute*_contact_distance`, and second – `*attribute*_contact_undecidable_range`, where **attribute** will be replaced by class `monomer_hallmark` attribute.

Similarly for `SetDistanceCriterion` one needs to define names of mers attributes that are lists or dicts of atoms to be searched for:

```
from pydesc.contacts import PointsDistanceCriterion

class PaACrt(SetDistanceCriterion):
    monomer_hallmark = 'pseudoatoms'
    monomer_hallmark2 = 'atoms'

crt = PaACrt(criterion_distance=4.5, undecidable_range=.5,
    num_of_checked_pairs=2)
```

Initializator of such criteria can be feed with threshold and undecidable range, but also with number of pairs to meet those criteria. By default it is set to 2.

## 9.4 Defining own mers, pseudoatoms and contact criteria

By deafult PyDeck works on all-atom representation and is meant to be a tool in protein and nucleic acid structures analysis, but can be also used for other representations and for other molecules. In both cases it is more convenient to define new kinds of mers. Mers in PyDesc are stored in `pydesc.monomer` module. Main class, `Monomer` is abstract and has two subclasses: `MonomerChainable` and `MonomerOther`, first of which is, again, an abstract superclass for `Nucleotide` and `Residue` classes, while later is superclass for `Ligand` and `Ion`. Lets assume we have some `pdb` file containing protein structure, but in representation containing only carbons alpha, beta and some pseudoatoms representing side chain, called *SSn*, where $n$ is number of that pseudoatom (e.g. MARTINI representation). As that representation lacks important backbone atoms – by default PyDesc would fail to load it, unless new subclass of `MonomerChainable` is defined. We will be interesed in, lets say, contact map of that protein with contact criterion defined as distance between *SS1* pseudoatoms, but only for serines. Therefore we need:

```
from pydesc.monomer import MonomerChainable
from pydesc.monomer import Monomer

class ReducedResidue(MonomerChainable):
    def __init__(self, pdb_residue, structure_obj, **kwargs):
        Monomer.__init__(self, pdb_residue, structure_obj,
            **kwargs)
    @property
    def ss1(self):
        return self.atoms['SS1']
```

As you can notice, this class simply calls `Monomer` initializator (instead of `MonomerChainable` to avoid questions about backbone atoms) and defines a new property, `ss1`, which returns **atom** called *SS1*. In PyDesc everything that is read from `pdb` file is atom and to make it accessible for contact criteria – one needs to define a property returning that atom.

Now movin og to contact criteria. As mentioned before, Users criterion contact should be subclass of classes provided in `pydesc.contacts` module. Usually contact criteria are classes that use two methods to tell if two mers are in contact or not: `_calculate_distance` and `_is_in_contact`. Both should take two mer instances as arguments. First one returns distance or array of distances, dependent on criterion type. Second one should call first one and compare results obtained for given mers with thresholds. For this case we should write:

```
from pydesc.contacts import ContactCriterion

class SerSS1Crt(ContactCriterion):
    def __init__(self, threshold, undecidable_range):
        self.thr = threshold
        self.undr = undecidable_range

    def _calculate_distance(self, m1, m2, **kwargs):
        return (m1.ss1 - m2.ss2).calculate_length()

    def _is_in_contact(self, m1, m2, **kwargs):
        if m1.name != m2.name != 'SER':
            return 0
        dist = self._calculate_distance(m1, m2)
        if dist < self.thr - self.undr:
            return 2
        elif dist < self.thr + self.undr:
            return 1
        return 0
```

With both those classes we are ready to load file with such reduced representation and calculate contact map for criterion concerning *SS1* pseudoatoms in serines only. Structure in that representation is provided as `IOStream` (file-like object) within `pydesc.sample` module as result of `get_MARTINI_structure`, so to test it use:

```
from pydesc.sample import get_MARTINI_structure

s = get_MARTINI_structure()
s.set_contact_map(SerSS1Crt(5.0, 0.5))
```

## 10 DLS: descriptors of local structure

That chapter will describe advanced usage of descriptors and is under construction.

## 11 Structure comparison

Every module presented so far is written in Python, therefore is easy to modify and extend. As structure comparison is computationally demanding, to speed up process – parts responsible for that feature in PyDesc are written in `C` and not available for modifications.

All modules are availible in `pydesc.cydesc` subpackage. There are three main modules: `overfit` for calculation of superposition of already aligned structures, `fitdesc` for finding patterns in structures and `compdesc` for descriptor comparison.

### 11.1 OverFit

Module for structure superposition calculation. Lets show how it works on an example: in aminoacyl-tRNA structure Rossmann fold is present. Corresponding substructures in `1n3l`, residues A181-A212 and `2dlc`, residues X285-X216 slightly differ in sequence, but show high similarity in structure. To calculate RMSD and rotation matrix for their superposistion with PyDesc simply write:

```
from pydesc.structure import StructureLoader
from pydesc.cydesc.overfit import Overfit

stc_1n3l = StructureLoader().load_structure('1n3l')[0]
stc_2dlc = StructureLoader().load_structure('2dlc')[0]

ovf = Overfit()
ovf.add_structure(stc_1n3l['A181': 'A212'], stc_2dlc['X185':
    'X216'])
results = ovf.overfit()

print results
```

Output:

```
(0.47124961018562317, <pydesc.geometry.TRTMatrix object at
    0x7f4ab52eda90>)
```

Method `overfit` returns tuple of two objects: floating number, which is RMSD and PyDesc `TRTMatrix`, which can be applied to transform objects. Using this matrix to transform `1n3l` will superimpose it to `2dlc`:

```
stc_1n3l.trt_matrix = stc_1n3l.trt_matrix.combine(results[1])
```

Comarping multiple structures is possible thanks to `Multifit` object, for which principles are the same:

```
stc_5thl = StructureLoader().load_structure('5thl')[0]

mft = Multifit()
mft.add_structure(stc_1n3l['A181': 'A212'], stc_2dlc['X185':
    'X216'], stc_5thl['A181': 'A212'])
results = mft.multifit()
```

Except that for this object result is a list of pairs: RMST and rotation matrix, that could be applied to subsequent objects passed to `add_structure` method.

The whole procedure can be shorten by simple call of function `overfit` from that module:

```
from pydesc.cydesc.overfit import overfit

results = overfit(stc_1n3l['A181': 'A212'], stc_2dlc['X185':
    'X216'])
```

The same works for `multifit` function.

## 11.2 CompDesc

Compdesc is used to compare descriptors defined as in chapter **??** and obtaines as described in chapter **??**. For two structures loaded in previous example, lets compare all possible descriptors. First we create them:

```
from pydesc.structure import ProteinDescriptor
from pydesc.cydesc.compdesc import CompDesc

stc_1n3l.set_contact_map()
stc_2dlc.set_contact_map()

dscs_1n3l = [i for i in
    ProteinDescriptor.create_descriptors(stc_1n3l) if
    isinstance(i, ProteinDescriptor)]
dscs_2dlc = [i for i in
    ProteinDescriptor.create_descriptors(stc_2dlc) if
    isinstance(i, ProteinDescriptor)]

results = []
for i in dscs_1n3l:
    row = []
    for j in dscs_2dlc:
        row.append(CompDesc(i, j).compdesc())
```

Each results contains three elements: RMAS, pair alignment object (see chapter **??**) and rotation matrix, as it was in case of overfit.

## 11.3 FitDesc

`FitDesc` from `fitdesc` module allows us to find structural pattern in greater structure. Lets use it to find part of Rossmann fold from structure `1n3l` mentioned in `overfit` section in `2dlc`:

```
from pydesc.cydesc.fitdesc import FitDesc

fit = FitDesc(stc_1n3l['A181': 'A212'], stc_2dlc)
results = fit.fitdesc(2.5, 0)

print results
```

`Fitdesc.fitdesc` method takes two arguments: maximal RMSD of superimposition of motiff to structure and maximal number of results, which is unlimited if set to 0. Again, function `fitdesc` makes everything easier:

```
from pydesc.cydesc.fitdesc import fitdesc

results = fitdesc(stc_1n3l['A181': 'A212'], stc_2dlc, 2.5)

print results
```

As a results one gets list of triple tuples containing RMSD, alignment object and rotation matrix.

## 12 Trajectories

That chapter will describe work with `dcd` trajectories in PyDesc and is under construction.

## 13 Alignments

That chapter will describe how to deal with alignment files in PyDesc and is under construction.