Pyidp3 Documentation

Release 0.0.5

Simon Vandevelde

May 31, 2022

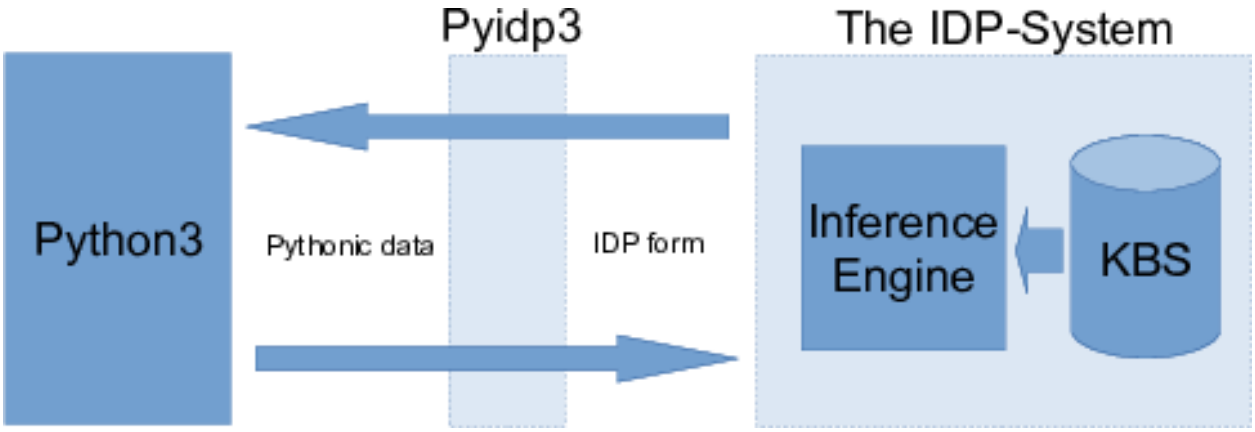
Contents:

This documentation covers everything in the Pyidp3 module.

This is a Python3 port of Joost Vennekens Pyidp.

The Pyidp3 module is an API between Python3 and the IDP system. In short, IDP is a Knowledge Base System (KBS) using the FO(.) language. FO(.) is standard First-Order logic, but expanded. See the IDP website for more. A KBS is a system that stores all it's knowledge in a knowledge base, and then supports different inference methods to apply on the knowledge. It's programmed in a declarative manner. More on programming the IDP system and FO(.) can be found here.

Pyidp3 will try to bridge the gap between IDP (which is programmed *declaratively*) and Python (which is programmed *imperatively*). It works in both directions: the user can supply data in Pythonic form to Pyidp3, which will then be converted to IDP form and given to the IDP system. When the IDP system is done infering, Pyidp3 will process it's output and translate this back into Pythonic form.



A list of all the features can be found at: *Pyidp3 features*.

More information on the porting of Pyidp to Pyidp3 can be found at: *Porting of Pyidp to Pyidp3*.

This submodule is part of my master's thesis. Due to timeconstraints, this module is far from perfect. I used it to build an application to assign students to groups, based on the IDP system. More on that can be found here.

CHAPTER 1

Pyidp3 features

Because Pyidp3 is a port of Pyidp, not all features were added by me.

1.1 Existing features from Pyidp

Here is the list of features that were already in Joost Vennekens' Pyidp (and were merely ported by me):

- Parsing Python to IDP, and from IDP to Python.
- Converting Pythonic to IDP-form. *(This is no longer supported in Pyidp3. The code is there, so it might work but I didn't do any active development on it.)
- Support for *Type*, *Predicate*, *Function*, *Constant* and *Definitions*.
- Support for *vocabulary*, *theory* and *structure*.
- Basic model expansion.

1.2 Added features in Pyidp3

Here is the list of features added in Pyidp3:

- Sphinx documentation (you're reading it!).
- Documentation throughout the code, to make it more readable.
- Support for adding the *Term* block, as a subclass of *Block*.
- Support for **constructed_from** keyword in a *Type*.
- Support for **isa** keyword in a *Type*.
- Model expansion is now done by calling *.model_expand()*.
- Implemented a way to minimize, by adding the *.minimize(term)* method.

- Implemented a way to SATcheck, by adding the *.sat_check()* method.
- Users can now also set IDP options (All options! Most of them haven't been tested though).
- The *model_expand* and *minimize* methods are now able to return multiple solutions, instead of only one.
- The IDP object now has a *compare* method to compare two enumerables and list the differences (currently only for dictionaries).

1.3 Added QOL in Pyidp3:

Along with features, some 'Quality-Of-Life' related functionalities were added to Pyidp3. These are all features that aren't completely *necessary*, but are just nice to have (and improved my QOL as the maintainer of this module).

- Changed the structure of the .py files to an actual module.

When porting I deleted the __init__.py file because I didn't know what it did. I imported all the submodules relatively, which was a massive pain.

- Added PEP8 conformity.

This is mostly codelines being longer than 79 characters, or forgetting whitespace. These don't break functionality, but it makes everything harder to read!

- Automated testing!

Using GitLab's free CI and a custom-made Dockerfile, I was able to automate my testing. Thanks to this I was able to find a lot of bugs I had written before I pulled my code into the masterbranch. Thanks GitLab!

- Written a .idp to .py converter.

During my testing I needed a lot of testfiles, and creating them all by hand took a while. Which is why I tried my best at creating a .idp to .py converter. As of right now it mostly works, there's just some minor kinks I need to get out before I can call it actually done.

- Pyidp3 doesn't add a '.' to a constraint that already contains one at the end.

Before I constantly forgot that Pyidp3 automatically adds a '.' at the end of a constraint. Which means that if I manually added a '.', I would have two dots at the end of the line, and IDP wouldn't be able to interpret it. Now it checks whether it needs to add a '.' before actually adding one.

1.4 Fixed bugs:

The original Pyidp version also had some bugs. However, I do suspect most of them were introduced by me when I first ported Pyidp to Pyidp3. Still, here's a list of all the bugs that were fixed.

- Fixed a bug in the *Constraint* method, where it didn't use self.know but appended directly to itself.
- *IDPIntTypes* are no longer generated number by number.

```
foo = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10} //old
foo = {0..10} //new
```

- 'string' wasn't being recognized as a possible type.
- String had to be supplied with extra quotation marks.

```
Type(example, {1: '"first"', 2: '"second"'})
```

would be needed to translate to

```
example = {1->"first"; 2->"second"}
```

Now this is no longer needed, and leaving the outer quotation marks is possible.

```
Type(example, {1: "first", 2: "second"})
```

- Definitions are no longer formed incorrectly.
- Fixed the CI testing platform not having support for unicode utf-8 characters.
- Constants can now get a value:

```
Constant("Apple", 5)
```

will now result in:

```
Apple: 5
```

in the theory.

- 'nat' wasn't begin recognized as a possible type.

CHAPTER 2

Basic Tutorial

Because this module can be somewhat daunting at times, here is a (hopefully) comprehensible tutorial of all that it has to offer.

If this still isn't enough, or it's missing something, let me know. Until then, you could read through the *Examples using the Pyidp3 API* to see if it's covered there. And if it isn't, check out the *The Pyidp3 API reference*.

2.1 Requirements

IDP:

The IDP system needs to be installed somewhere on your system. To do this, go to this link, and download the offline version.

Open it using a filemanager or by using the "tar" command in terminal.

```
tar -xf ipd-version-tar.gz
```

This will extract the files from the tarball archive.

You can move the idp folder to anywhere you like, as long as the permissions are correct. I usually place my personal installation in my $HOME folder.

Pyidp3: Next up we need the Pyidp3 module. This can be done easily by using the following command:

```
python3 pip install pyidp3
```

If you used this command, proceed to the next step.

You could also download the sourcecode over at gitlab and clone it to your machine.

Next, you can run

```
python3 setup.py install
```

to install the package globally.

2.2 Tutorial

With the requirements done, let's start simple: to begin programming using Pyidp3 you need to import it. This is done as such:

```
from pyidp3.typedIDP import IDP
```

As a user, all you need is this import. The IDP object is the most top-level object there is. It will allow you to do everything you need to do.

This is a list of what the IDP object offers:

- **Constant**
- **Constraint**
- **Define**
- **Function**
- **Predicate**
- **Type**
- **check_sat**
- **minimize**
- **model_expand**

Say for instance, we'd like implement the following .idp file as a .py file, to be used in an application later. This example solves a simple letterpuzzle, where we try to assign values to each letter, so "AI + BA = CDE". No two letters can have the same value, none of the letters can be zero and all vowels need to be an even number, all consonants uneven.

```
vocabulary V {
    type Decimal isa int
        A: Decimal
    B: Decimal
    C: Decimal
    I: Decimal
    D: Decimal
    E: Decimal

    Even(Decimal)
}
structure S : V {
    Decimal = {0..9}
    Even = {0;2;4;6;8;}
}
theory T : V {
        A ~= B & A ~= C & A ~= I & A ~= D & A ~= E & B ~= C & B ~= I
        & B ~= D & B ~= E & C ~= I & C ~= D & C ~= E & I ~= D & I ~= E
        & D ~= E.

    A ~= 0.
    B ~= 0.
    C ~= 0.

    Even(A).
    Even(E).
```

(continues on next page)

(continued from previous page)

```
    Even(I).

    ~Even(B).
    ~Even(C).
    ~Even(D).

    I + A + 10*A + 10*B = E + 10 * D + 100 * C.
}
procedure main() {
    stdoptions.nbmodels = 3
    printmodels(modelexpand(T, S))
}
```

First of all, we need to instantiate our IDP object. This can be done as follows:

```
idp = IDP('path/to/IDP/exec')
```

Make sure to set the path to your IDP executable, or it won't work.

Next up, we'll add the type *Decimal*.

```
idp.Type("Decimal", (0, 9))
```

Normally, when the Type uses the **isa** keyword, we'd have to explicitly tell this to the Type() method. But, by default Pyidp3 will find if it's an **int** and add this automatically.

Now we should add the constants. This is also fairly easy:

```
idp.Constant("A: Decimal")
idp.Constant("B: Decimal")
idp.Constant("C: Decimal")
idp.Constant("I: Decimal")
idp.Constant("D: Decimal")
idp.Constant("E: Decimal")
```

Now for the Predicate *Even*, where we define what numbers are even.

```
idp.Predicate("Even(Decimal)", [0, 2, 4, 6, 8])
```

All that's left now are the constraints in our *theory*. Because these are already in their IDP form, we need to set the last variable to **True**. These are also fairly easy to create, but take some time to type out:

```
# No two letters can be the same.
idp.Constraint("A ~= B & A ~= C & A ~= I & A ~= D & A ~= E & B ~= C & B ~= I
                & B ~= D & B ~= E & C ~= I & C ~= D & C ~= E & I ~= D & I ~= E
                & D ~= E.", True)

# No letter can be zero.
idp.Constraint("A ~= 0.", True)
idp.Constraint("B ~= 0.", True)
idp.Constraint("C ~= 0.", True)
idp.Constraint("I ~= 0.", True)
idp.Constraint("D ~= 0.", True)
idp.Constraint("E ~= 0.", True)

# Vowels have to be an even number.
idp.Constraint("Even(A).", True)
```

(continues on next page)

(continued from previous page)

```
idp.Constraint("Even(E).", True)
idp.Constraint("Even(I).", True)

# Consonants have to be an uneven number.
idp.Constraint("~Even(B).", True)
idp.Constraint("~Even(C).", True)
idp.Constraint("~Even(D).", True)

# The formula to solve.
idp.Constraint("I + A + 10*A + 10*B = E + 10 * D + 100 * C.", True)
idp.Constraint("satisfiable()", True)
```

Note how we add **satisfiable()** as a constraint: otherwise IDP might return models that aren't satisfiable. This has it's applications, but for now we only want models that are satisfiable.

But say we'd like to extend this puzzle, with more letters and a harder to solve formula. This would be a pain, because for each n'th letter we'd have to add n-1 + 1 + 1 constraints. This is where the power of Pyidp3 (and Python) comes in. Because this is Python, we could also write something along the lines of:

```
idp.Type("Decimal", (0,9))
idp.Predicate("Even(Decimal)", [0, 2, 4, 6, 8])

even_letters = ['A', 'E', 'I']
uneven_letters = ['B', 'C', 'D']
letters = even_letters + uneven_letters
for i, letter in enumerate(letters):
    idp.Constant(letter+": Decimal")  # Define a type for every letter.

    for j, letter2 in enumerate(letters):  # Iterate over all the letters again
        if i < j and letter != letter2:  # No letters can have the same value
            idp.Constraint(letter + " ~= " + letter2)

    if letter in even_letters:
        idp.Constraint("Even(" + letter + ")")
    else:
        idp.Constraint("~Even(" + letter + ")")

    idp.Constraint(letter + " ~= 0")

idp.Constraint("I + A + 10*A + 10*B = E + 10 * D + 100 * C.", True)
idp.Constraint("satisfiable()", True)
```

Now if we want to add letters, all we need to do is append them to either the *even_letters* list or the *uneven_letters* list, and Pyidp3 will sort out the rest. In other words, our implementation is a lot more scalable than a direct implementation in IDP.

Last but not least, we still need to model expand! In the *main* block, *stdoptions.nbmodels* is set to three, after which it model expands. This can be done like so:

```
idp.nbmodels = 3
solutions = idp.model_expand()

print("Total amount of solutions:", len(solutions))
for i, sol in solutions:
    print("Solution {:d}".format(i), sol)
```

Every option in the IDP system can be used by giving a value to *IDP.optionname*. For a list of these options, see the

usermanual. One thing to note however, is that all the verbosityoptions need to be used as *IDP.verbosity_optionname*.

When there are multiple solutions and a lot of data, it can sometimes be hard to see the difference between two solutions. To help remedy this, you could use the *compare* function in the *IDP* class. Although currently, this only works on *functions*.

This is just the tip of the iceberg, but currently I don't have the time to add more to this tutorial. Luckily, there's a whole bunch of examplefiles you can read and try to reverse-engineer from. These can be found here: *Examples using the Pyidp3 API*.

CHAPTER 3

Examples using the Pyidp3 API

All of these examples were made to be used in my CI testing pipeline. For each of the examples, I list up what elements are in it in case you're looking for an example of something specific.

3.1 simple_inference

As the title says, this is a simple inference test. Contains:

- **nbmodels** and **xdb** options
- Defining *constants* without value or type
- *Constraint*
- Satchecking
- Model expanding

```
#!/usr/bin/python3

"""
This is a very simple and small test.
If this test fails, there's something extremely wrong.
It defines 3 variables, that each are either True or False.
"""
from pathlib import Path
from pyidp3.typedIDP import IDP


home = str(Path.home())
idp = IDP(home+"/idp/usr/local/bin/idp")
idp.nbmodels = 2
idp.xsb = "true"

idp.Constant("P")
```

(continues on next page)

(continued from previous page)

```
idp.Constant("Q")
idp.Constant("R")

idp.Constraint("(P => Q) <=> (P <= R).", True)
idp.Constraint("satisfiable().", True)

idp.check_sat()
idp.model_expand()
```

3.2 harder_inference

This is a bit tougher than the previous example. Contains:

- *Type* with a range
- *Constant* with type, no value
- *Constraint*
- *Predicate*
- Satchecking
- Model expanding

```
#!/usr/bin/python3

"""
This is a fairly simple test, it just adds more constants and constraints.
The generated IDP file will have 6 letters (each an int),
which have some constraints (A, E and I need to be even numbers,
while B, C and D need to be uneven).
A, B and C also can't be zero.
No two letters can be the same number.

The following puzzle needs to be solved:
    AI
    BA
+-----
   CDE

Or in other terms: AI + BA = CDE
"""
from pathlib import Path
from pyidp3.typedIDP import *


home = str(Path.home())
idp = IDP(home+"/idp/usr/local/bin/idp")

# Define the range of numbers
idp.Type("Decimal", (0, 9))

# Define the letters
```

(continues on next page)

(continued from previous page)

```
letters = ['A', 'B', 'C', 'I', 'D', 'E']
even_letters = ['A', 'I', 'E']
for letter in letters:
    idp.Constant(letter+": Decimal")
    for letter2 in letters:
        if letter == letter2:
            continue
        idp.Constraint(letter + " ~= " + letter2, True)
    if letter in even_letters:
        idp.Constraint("Even(" + letter + ")", True)
    else:
        idp.Constraint("~Even(" + letter + ")", True)


# Define the Even numbers
idp.Predicate("Even(Decimal)", [0, 2, 4, 6, 8])

# No two letters can have the same value
idp.Constraint("A ~= B & A ~= C & A ~= I & A ~= D & A ~= E & B ~= C & B ~= I"
               "& B ~= D & B ~= E & C ~= I & C ~= D & C ~= E & I ~= D &"
               " I ~= E & D ~= E", True)
idp.Constraint("I + A + 10*A + 10*B = E + 10 * D + 100 * C", True)

idp.check_sat()
idp.model_expand()
```

3.3 definition_test

This is a graphsolver with a Define in it. Other than that, nothing special going on. Contains:

- *Type*
- *Predicate*, with and without value
- *Define*
- Satchecking
- Modelexpansion

```
#!/usr/bin/python3

"""
This testfile is a graph solver.
Given a set of edges, it finds out what nodes are connected.
To do this is uses a definition consisting of two rules.

"""
from pathlib import Path
from pyidp3.typedIDP import *


home = str(Path.home())
```

(continues on next page)

(continued from previous page)

```
idp = IDP(home+"/idp/usr/local/bin/idp")

# Define Node, Edge and Connected
nodelist = ['A', 'B', 'C', 'D', 'E']
idp.Type("Node", nodelist)
idp.Predicate("Edge(Node, Node)", ["A,B", "A,C", "B,E", "C,D", "C,E", "D,E",
                                   "E,B"])
idp.Predicate("Connected(Node, Node)")

idp.Define("!x[Node] y[Node]: Connected(x,y) <- Edge(y,x). \n"
           "!x[Node] y[Node] z[Node]: Connected(x,y) "
           "<- Connected(x,z) & Connected(z,y)", True)

idp.check_sat()
idp.model_expand()
```

3.4 constructed_from

This file was mainly made to test the **constructed from** keyword. It contains:

- *Type*, with **constructed from**
- *Predicate*, with value
- *Constraint*
- Satchecking
- Modelexpansion

```
#!/usr/bin/python3

"""
Tests for "constructed_from".
This .idp file can only work when "constructed_from" works.
"""
from pathlib import Path
from pyidp3.typedIDP import *


home = str(Path.home())
idp = IDP(home+"/idp/usr/local/bin/idp")

days_of_the_week = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
                    "Saturday", "Sunday"]
idp.Type("Day", days_of_the_week, constructed_from=True)
idp.Predicate("Weekend(Day)", ["Saturday", "Sunday"])
idp.Constant("Easter: Day")

idp.Constraint("Weekend(Easter)", True)
idp.Constraint("Easter ~= Saturday", True)

idp.check_sat()
```

(continues on next page)

(continued from previous page)

```
idp.model_expand()
```

3.5 initial_group_assign

This file is part of my master's thesis. It's a simplified version of my initial group assign, where the IDP-system tries to assign students to groups as best as it can. It tries to minimize a term "Totaal". This currently isn't in English, for which I'm sorry.

It contains:

- **mxtimeout** option
- *Types*, with lists as values
- *Predicate*
- *Function*
- *Constant*
- *Define*
- *Constraint*
- Satchecking
- Modelexpansion
- Minimization, using the *Totaal* term.

```
#!/usr/bin/python3

"""
Testfile to test if the initial solution to our problem still works as it
should be.
It minimizes, and then it modelexpands.
"""
from pathlib import Path
from pyidp3.typedIDP import *


home = str(Path.home())
idp = IDP(home+"/idp/usr/local/bin/idp")
idp.mxtimeout = 10

WoonDict = {1: 87, 2: 98, 3: 80, 4: 80, 5: 83, 6: 88, 7: 90, 8: 86, 9: 80, 10:
            53, 11: 98, 12: 57, 13: 86, 14: 82, 15: 98, 16: 83, 17: 15, 18: 88,
            19: 22, 20: 78, 21: 80, 22: 86, 23: 27, 24: 57, 25: 84}
ZoneDict = {1: 2, 2: 1, 3: 2, 4: 2, 5: 2, 6: 2, 7: 2, 8: 2, 9: 2, 10: 2, 11: 1,
            12: 2, 13: 2, 14: 2, 15: 1, 16: 2, 17: 2, 18: 1, 19: 2, 20: 1, 21:
            2, 22: 2, 23: 2, 24: 2, 25: 2}

idp.Type("Student", list(range(1, 26)))
idp.Type("Getal", list(range(int(1000000))))
idp.Type("Postcode", list(range(0, 100)))
```

(continues on next page)

(continued from previous page)

```
idp.Type("Zone", list(range(0, 10)))

idp.Predicate("Samen(Student,Student)")
idp.Predicate("VolSamen(Student, Student)")

idp.Function("Woont(Student): Postcode", WoonDict)
idp.Function("WoontZone(Student): Zone", ZoneDict)

idp.Predicate("Wortel(Student)")
idp.Predicate("Blad(Student)")
idp.Function("Aant(Student): Getal")

idp.Constant("SamenSchool: Getal")
idp.Constant("Afstand: Getal")
idp.Constant("UitZone: Getal")
idp.Constant("Totaal: Getal")


idp.Define("Wortel(x) <- x < min{y[Student]: Samen(y,x):y}.", True)
idp.Define("Blad(x) <- ~Wortel(x).", True)

idp.Constraint("#{x[Student]: Wortel(x)} = 5", True)
idp.Constraint("!x[Student]: Blad(x) <=> ?y[Student]: Wortel(y) & Samen(y,x)",
               True)
idp.Constraint("!x[Student]: Aant(x) = #{y[Student]: Samen(x,y) | Samen(y,x)}",
               True)
idp.Constraint("!x[Student]: Wortel(x) <=> 4 =< Aant(x) =< 6", True)
idp.Constraint("!x[Student]: Blad(x) <=> Aant(x) = 1", True)
idp.Constraint("!x[Student] y[Student]: Samen(x,y) => Wortel(x) & Blad(y)",
               True)
idp.Define("!x[Student] y[Student] z[Student]: VolSamen(y,z) <- Wortel(x) & y"
           " < z & Samen(x,y) & Samen(x,z).\n"
           "!x[Student] y[Student]: VolSamen(x,y) <- Wortel(x) & Samen(x,y).",
           True)
idp.Constraint("Afstand = sum{x[Student] y[Student]: x < y & VolSamen(x,y) &"
               " WoontZone(x) = WoontZone(y): abs(Woont(x) - Woont(y))}", True)
idp.Constraint("UitZone = #{x[Student] y[Student]: x < y & VolSamen(x,y) &"
               " WoontZone(x) ~= WoontZone(y) }", True)

idp.Constraint("Totaal = Afstand  + UitZone * 100", True)
idp.check_sat()
idp.model_expand()
sols = idp.minimize("Totaal")


for sol in sols:
    print(sol)
    if sol['satisfiable']:
        for x in sol['Samen']:
            print(x)
```

3.6 initial_group_assign_plus

This file is an expanded version of the previous, where schools are taken into account.

It contains:

- **mxtimeout** and **nbmodels** option
- *Types*, with lists as values
- *Predicate*
- *Function*
- *Constant*
- *Define*
- *Constraint*
- Satchecking
- Modelexpansion
- Minimization, using the *Totaal* term.

```
#!/usr/bin/python3

"""
Testfile that checks whether ISSUE-1 (the {x..y} in structs) is still valid.
"""
from pathlib import Path
from pyidp3.typedIDP import *


home = str(Path.home())
idp = IDP(home+"/idp/usr/local/bin/idp")
idp.mxtimeout = 10
idp.nbmodels = 5

WoonDict = {1: 87, 2: 98, 3: 80, 4: 80, 5: 83, 6: 88, 7: 90, 8: 86, 9: 80, 10:
            53, 11: 98, 12: 57, 13: 86, 14: 82, 15: 98, 16: 83, 17: 15, 18: 88,
            19: 22, 20: 78, 21: 80, 22: 86, 23: 27, 24: 57, 25: 84}
ZoneDict = {1: 2, 2: 1, 3: 2, 4: 2, 5: 2, 6: 2, 7: 2, 8: 2, 9: 2, 10: 2, 11: 1,
            12: 2, 13: 2, 14: 2, 15: 1, 16: 2, 17: 2, 18: 1, 19: 2, 20: 1, 21:
            2, 22: 2, 23: 2, 24: 2, 25: 2}
SchoolDict = {1: "GO! atheneum Anderlecht",
              2: "KONINKLIJK ATHENEUM KOEKELBERG",
              3: "TSM-Bovenbouw"', 4: '"TSM-Bovenbouw",
              5: "PTS, Provinciale Scholen voor Tuinbouw en Techniek",
              6: "Onze-Lieve-Vrouw-Presentatie", 7: "Sint-Ludgardisschool",
              8: "Mater Salvatorisinstituut", 9: "Scheppersinstituut",
              10: "Sint-Gabriëlcollege", 11: "Scheppersinstituut",
              12: "Sint-Norbertusinstituut 2", 13: "Ursulinen Mechelen 1",
              14: "College Hagelstein 2", 15: "Scheppersinstituut",
              16: "TSM-Bovenbouw", 17: "TOEKOMSTONDERWIJS HOBOKEN",
              18: "Gemeentelijk Technisch Instituut",
              19: "Heilig Hart - Bovenbouw 1",
              20: "Gemeentelijke Technische en Beroepsschool",
              21: "GO! Busleyden Atheneum-campus Pitzemburg",
              22: "College Hagelstein 2",
```

(continues on next page)

(continued from previous page)

```
              23: "Kardinaal van Roey-Instituut ASO",
              24: "Sint-Gummaruscollege", 25: "GO! atheneum Boom"}


idp.Type("Student", (1, 25))
idp.Type("Getal", (0, 10000000))
idp.Type("Postcode", (0, 100))
idp.Type("Zone", (0, 10))

idp.Predicate("Samen(Student,Student)")
idp.Predicate("VolSamen(Student, Student)")

idp.Function("Woont(Student): Postcode", WoonDict)
idp.Function("WoontZone(Student): Zone", ZoneDict)
idp.Function("School(Student): string", SchoolDict)

idp.Predicate("Wortel(Student)")
idp.Predicate("Blad(Student)")
idp.Function("Aant(Student): Getal")

idp.Constant("SamenSchool: Getal")
idp.Constant("Afstand: Getal")
idp.Constant("UitZone: Getal")
idp.Constant("Totaal: Getal")


idp.Define("Wortel(x) <- x < min{y[Student]: Samen(y,x):y}.", True)
idp.Define("Blad(x) <- ~Wortel(x).", True)

idp.Constraint("#{x[Student]: Wortel(x)} = 5", True)
idp.Constraint(" !x[Student]: Blad(x) <=> ?y[Student]: Wortel(y) & Samen(y,x)",
               True)
idp.Constraint("!x[Student]: Aant(x) = #{y[Student]: Samen(x,y) | Samen(y,x)}",
               True)
idp.Constraint("!x[Student]: Wortel(x) <=> 4 =< Aant(x) =< 6", True)
idp.Constraint("!x[Student]: Blad(x) <=> Aant(x) = 1", True)
idp.Constraint("!x[Student] y[Student]: Samen(x,y) => Wortel(x) & Blad(y)",
               True)
idp.Define("!x[Student] y[Student] z[Student]: VolSamen(y,z) <- Wortel(x) & y"
           " < z & Samen(x,y) & Samen(x,z).\n"
           "!x[Student] y[Student]: VolSamen(x,y) <- Wortel(x) & Samen(x,y).",
           True)
idp.Constraint("SamenSchool = #{x[Student] y[Student]: x < y & VolSamen(x,y)"
               "& School(x) ~= School(y)}", True)
idp.Constraint("Afstand = sum{x[Student] y[Student]: x < y & VolSamen(x,y) &"
               " WoontZone(x) = WoontZone(y): abs(Woont(x) - Woont(y))}", True)
idp.Constraint("UitZone = #{x[Student] y[Student]: x < y & VolSamen(x,y) &"
               " WoontZone(x) ~= WoontZone(y) }", True)

idp.Constraint("Totaal = Afstand  + UitZone * 100 + SamenSchool*100", True)


solutions = idp.model_expand()
solutions = idp.minimize("Totaal")
idp.check_sat()

print("{:d} solutions!".format(len(solutions)))
```

(continues on next page)

(continued from previous page)

```
for index, sol in enumerate(solutions):
    if sol['satisfiable']:
        print("Sol {:d}:".format(index))
        for x in sol['Samen']:
            print('\t', x)
    else:
        continue
```

3.7 further_group_assign

This is the other part of my master's thesis. Here we try to reassign students based on their old groups, and a preference.

It contains:

- **nbmodels** option
- comparing solutions using the *compare* method
- *Type*, also one with the **isa** keyword
- *Constant*, with value
- *Predicate*
- *Function*
- *Constraint*
- *Define*
- minimization, using the *Total* term

```
#!/usr/bin/python3

"""
Testfile which attempts to assign students to a group, based on a previous
group and a preference.
"""
from pathlib import Path
from pyidp3.typedIDP import *


home = str(Path.home())
idp = IDP(home+"/idp/usr/local/bin/idp")
idp.nbmodels = 5

# All the inputvars
idp.Type("Student", (1, 152))
idp.Type("Number", (0, 10000000))
idp.Type("Group", (2, 11))
idp.Type("Priority", "Number", isa=True)
idp.Constant("MinSize: Number", 14)
idp.Constant("MaxSize: Number", 16)

together = [(142, 139, 2), (139, 143, 1), (139, 142, 1)]
not_together = [(142, 143, 2), (142, 141, 2), (142, 140, 2)]
```

(continues on next page)

(continued from previous page)

```
in_group = []
group_dict = {1: 2,  2: 3,  3: 4,  4: 5,  5: 6,  6: 7,  7: 8,  8: 9,  9: 10,
              10: 11,  11: 3,  12: 11,  13: 8,  14: 6,  15: 4,  16: 2,  17: 10,
              18: 8,  19: 7,  20: 5,  21: 4,  22: 3,  23: 2,  24: 11,  25: 9,
              26: 8, 27: 6,  28: 6,  29: 5,  30: 4,  31: 3,  32: 2,  33: 10,
              34: 11,  35: 9,  36: 5,  37: 7,  38: 7,  39: 6,  40: 5,  41: 5,
              42: 4,  43: 3,  44: 3,  45: 2,  46: 10,  47: 10,  48: 11,  49: 8,
              50: 9,  51: 7,  52: 8, 53: 10,  54: 6,  55: 6,  56: 5,  57: 5,
              58: 5,  59: 4,  60: 4,  61: 3, 62: 3,  63: 2,  64: 2,  65: 10,
              66: 7,  67: 10,  68: 11,  69: 11,  70: 9,  71: 9,  72: 8,  73: 8,
              74: 7,  75: 7,  76: 6,  77: 6,  78: 6,  79: 9,  80: 5,  81: 5,
              82: 4,  83: 4,  84: 4,  85: 3,  86: 3,  87: 3,  88: 2,  89: 2,
              90: 2,  91: 2,  92: 10,  93: 11,  94: 10,  95: 8,  96: 11,
              97: 11,  98: 9,  99: 9,  100: 9,  101: 9,  102: 8,  103: 8,
              104: 8, 105: 7,  106: 7,  107: 7,  108: 6,  109: 6,  110: 6,
              111: 6,  112: 5, 113: 5,  114: 5,  115: 5,  116: 4,  117: 4,
              118: 4,  119: 4,  120: 4, 121: 3,  122: 3,  123: 3,  124: 3,
              125: 2,  126: 2,  127: 2,  128: 2, 129: 11,  130: 10,  131: 10,
              132: 10,  133: 10,  134: 10,  135: 11, 136: 11,  137: 11,
              138: 11,  139: 9,  140: 9,  141: 9,  142: 9,  143: 9,  144: 8,
              145: 8,  146: 8,  147: 7,  148: 7,  149: 7,  150: 7,  151:
              7,  152: 6}
idp.Function("InGroup(Student): Group", group_dict)
idp.Predicate("WantsTogether(Student, Student, Priority)", together)
idp.Predicate("NotWantsTogether(Student, Student, Priority)", not_together)
idp.Predicate("WantsInGroup(Student, Group, Priority)", in_group)


# All the inner workings + output
idp.Function("GroupSize(Group): Number")

idp.Constant("Total: Number")
idp.Constant("TotUnsatTogether: Number")
idp.Constant("TotUnsatNotTogether: Number")
idp.Constant("TotUnsatInGroup: Number")
idp.Constant("TotUnsatNewGroup: Number")

idp.Predicate("UnsatTogether(Student, Student)")
idp.Predicate("UnsatNotTogether(Student, Student)")
idp.Predicate("UnsatInGroup(Student, Group)")
idp.Predicate("UnsatNewGroup(Student)")

idp.Function("NewInGroup(Student): Group")


# All the necessary constraints
idp.Constraint("!g[Group]: GroupSize(g) = #{x[Student]: NewInGroup(x) = g}",
               True)
idp.Constraint("!g[Group]: MinSize =< GroupSize(g) =< MaxSize", True)
idp.Constraint("TotUnsatNewGroup = #{x[Student]: InGroup(x) ~= NewInGroup(x)}",
               True)
idp.Constraint("TotUnsatTogether = sum{x[Student] y[Student]:"
               " WantsTogether(x,y,p) & NewInGroup(x) ~= NewInGroup(y): p}",
               True)
idp.Constraint("TotUnsatNotTogether = sum{x[Student] y[Student]:"
               " NotWantsTogether(x,y,p) & NewInGroup(x) = NewInGroup(y): p}",
               True)
```

(continues on next page)

(continued from previous page)

```
idp.Constraint("TotUnsatInGroup = sum{x[Student] g[Group]: WantsInGroup(x,g,p)"
               " & NewInGroup(x) ~= g: p}", True)
idp.Constraint("Total = TotUnsatTogether + TotUnsatNotTogether +"
               " TotUnsatInGroup + TotUnsatNewGroup", True)

idp.Define("!x[Student] y[Student]: UnsatTogether(x,y) <- WantsTogether(x,y,p)"
           " & NewInGroup(x) ~= NewInGroup(y).", True)
idp.Define("!x[Student] y[Student]: UnsatNotTogether(x,y) <- "
           " NotWantsTogether(x,y,p) & NewInGroup(x) = NewInGroup(y).",
           True)
idp.Define("!x[Student]: UnsatNewGroup(x) <- InGroup(x) ~= NewInGroup(x).",
           True)
idp.Define("!x[Student] g[Group]: UnsatInGroup(x,g) <- WantsInGroup(x,g,p) &"
           " NewInGroup(x) ~= g.", True)

sols = idp.minimize("Total")

newgroups = []
for sol in sols:
    newgroups.append(sol['NewInGroup'])
print(newgroups)
print("Verschil tussen sols:")
idp.compare(newgroups)
# idp.compare(sols)
```

3.8 sudoku

Sudokusolver.

Contains:

- **nbmodels** option
- *Type*
- *Constant*, with value
- *Predicate*
- *Function*
- *Constraint*
- *Define*
- Modelexpansion

```
#!/usr/bin/python3

"""
This testfile solves a sudoku.

"""
from pathlib import Path
from pyidp3.typedIDP import *
```

(continues on next page)

(continued from previous page)

```
home = str(Path.home())
idp = IDP(home+"/idp/usr/local/bin/idp")
idp.nbmodels = 10

idp.Type("Row", (1, 9))
idp.Type("Column", (1, 9))
idp.Type("Number", (1, 9))

start_numbers = [(1, 1, 8), (1, 1, 8),
                 (2, 3, 3), (2, 4, 6),
                 (3, 2, 7), (3, 5, 9), (3, 7, 2),
                 (4, 2, 5), (4, 6, 7),
                 (5, 5, 4), (5, 6, 5), (5, 7, 7),
                 (6, 4, 1), (6, 8, 3),
                 (7, 3, 1), (7, 8, 6),
                 (7, 9, 8),
                 (8, 3, 8),
                 (8, 4, 5),
                 (8, 8, 1),
                 (9, 2, 9),
                 (9, 7, 4)]
idp.Predicate("Square(Row, Column, Row, Column)")
idp.Predicate("Start(Row,Column,Number)", start_numbers)
idp.Predicate("Group(Row, Column, Row, Column)")
idp.Function("Solution(Row, Column): Number")

idp.Constraint("!r[Row] c[Column] number: Start(r, c, number) =>"
               "Solution(r, c) = number", True)
idp.Define("!r1[Row] c1[Column] r2[Row] c2[Column]:"
           "Square(r1,c1,r2,c2) <- r1- (r1-1)%3 = "
           " r2 - (r2-1)%3 & c1-(c1-1)%3 = c2-(c2-1)%3",
           True)
idp.Define("!r1[Row] c1[Column] r2[Row] c2[Column]: Group(r1, c1, r2, c2) "
           "<- Square(r1, c1, r2, c2).\n"
           "!r1[Row] r2[Row] c[Column]: Group(r1, c, r2, c) <- true.\n"
           "!r[Row] c1[Row] c2[Column]: Group(r, c1, r, c2).", True)
idp.Constraint("!r1[Row] c1[Column] r2[Row] c2[Column]: Group(r1, c1, r2, c2)"
               "& (r1 ~= r2 | c1 ~= c2) => Solution(r1,c1) ~="
               "Solution(r2,c2)", True)
idp.check_sat()
sols = idp.model_expand()
for index, sol in enumerate(sols):
    if sol['satisfiable']:
        print("Sol{:d}: ".format(index), sol['Solution'])
```

3.9 masyu

Contains:

- *Type*, **constructed_from**
- *Constant*, with value

- *Predicate*
- *Function*
- *Constraint*
- *Define*
- Satchecking
- Modelexpansion

```
#!/usr/bin/python3

"""
This is by far the testfile with the most in it (yet it's not the hardest!).
It solves a masyu puzzle.

"""
from pathlib import Path
from pyidp3.typedIDP import *


home = str(Path.home())
idp = IDP(home+"/idp/usr/local/bin/idp")

idp.Type("Row", (0, 4))
idp.Type("Column", (0, 4))
idp.Type("Pearl", ["Hollow", "Filled"], constructed_from=True)
idp.Type("Wire", ["NS", "EW", "ES", "WS", "NE", "NW", "Empty"],
         constructed_from=True)

pearl_position = [(0, 2, "Filled"),
                  (1, 4, "Filled"),
                  (2, 2, "Filled"),
                  (3, 0, "Filled"), (3, 1, "Filled"),
                  (4, 4, "Hollow")]
idp.Predicate("PearlPosition(Row, Column, Pearl)", pearl_position)
idp.Predicate("WireStraight(Wire)", ["NS", "EW"])
idp.Predicate("WireCurve(Wire)", ["ES", "WS", "NE", "NW"])
idp.Predicate("WireNorth(Wire)", ["NS", "NE", "NW"])
idp.Predicate("WireEast(Wire)", ["ES", "NE", "EW"])
idp.Predicate("WireSouth(Wire)", ["NS", "WS", "ES"])
idp.Predicate("WireWest(Wire)", ["EW", "WS", "NW"])
idp.Predicate("Link(Row, Column, Row, Column)")
idp.Predicate("Connected(Row, Column, Row, Column)")
idp.Function("Solution(Row, Column): Wire")

idp.Constraint("!r c: WireNorth(Solution(r,c)) => WireSouth(Solution(r-1,c))",
               True)
idp.Constraint("!r c: WireSouth(Solution(r,c)) => WireNorth(Solution(r+1,c))",
               True)
idp.Constraint("!r c: WireEast(Solution(r,c)) => WireWest(Solution(r,c+1))",
               True)
idp.Constraint("!r c: WireWest(Solution(r,c)) => WireEast(Solution(r,c-1))",
               True)
idp.Constraint("!r c parel: PearlPosition(r,c,parel) & parel = Hollow => "
               "WireCurve(Solution(r,c))", True)
idp.Constraint("!r c parel: PearlPosition(r,c,parel) & parel = Filled => "
               "WireStraight(Solution(r,c))", True)
```

(continues on next page)

(continued from previous page)

```
idp.Constraint("!r c parel: PearlPosition(r,c,parel) & parel = Hollow & "
               "WireNorth(Solution(r,c)) => WireStraight(Solution(r-1,c))",
               True)
idp.Constraint("!r c parel: PearlPosition(r,c,parel) & parel = Hollow & "
               "WireSouth(Solution(r,c)) => WireStraight(Solution(r+1,c))",
               True)
idp.Constraint("!r c parel: PearlPosition(r,c,parel) & parel = Hollow & "
               "WireEast(Solution(r,c)) => WireStraight(Solution(r,c+1))",
               True)
idp.Constraint("!r c parel: PearlPosition(r,c,parel) & parel = Hollow & "
               "WireWest(Solution(r,c)) => WireStraight(Solution(r,c-1))",
               True)
idp.Constraint("!r c parel: PearlPosition(r,c,parel) & parel = Filled & "
               "WireNorth(Solution(r,c)) => (WireCurve(Solution(r-1,c)) | "
               "WireCurve(Solution(r+1,c)))", True)
idp.Constraint("!r c parel: PearlPosition(r,c,parel) & parel = Filled & "
               "WireEast(Solution(r,c)) => (WireCurve(Solution(r,c-1)) | "
               "WireCurve(Solution(r,c+1)))", True)
idp.Constraint("!r1 c1 r2 c2: (Solution(r1,c1) ~= Empty & Solution(r2,c2) ~= "
               "Empty) => Connected(r1,c1,r2,c2)", True)
idp.Define("!r c: Link(r,c,r-1,c) <- WireNorth(Solution(r,c)) & "
           "WireSouth(Solution(r-1,c)). "
           "!r c: Link(r,c,r+1,c) <- WireSouth(Solution(r,c)) & "
           "WireNorth(Solution(r+1,c)). "
           "!r c: Link(r,c,r,c+1) <- WireEast(Solution(r,c)) & "
           "WireWest(Solution(r,c+1)). "
           "!r c: Link(r,c,r,c-1) <- WireWest(Solution(r,c)) &"
           "WireEast(Solution(r,c-1)).", True)

idp.Define("!r1 c1 r2 c2: Connected(r1,c1,r2,c2) <- Link(r1,c1,r2,c2). "
           "!r1 c1 r2 c2: Connected(r1,c1,r2,c2) <- ?r3 k3: "
           "Connected(r3,k3,r2,c2) & Connected(r1,c1,r3,k3). ", True)
idp.check_sat()
idp.model_expand()
```

CHAPTER 4

Porting of Pyidp to Pyidp3

- Firstly Python's own 2to3 was used for the initial port. This is a tool to *automagically* convert from Python2 to Python3. This produces a log of the changes made, which can be found at the bottom of this page.
- The way *popen* works was changed to only accept byte objects (and it doesn't convert them automatically), so I had make sure all input was encoded first, and all output was decoded afterwards. This can be done easily by using the *encode()* and *decode()* methods.

That's it. Using these two simple tricks, I was able to convert Pyidp to work on Python3 (*Doctors hate him!*). After porting, features were added, QOL was improved and bugs were squashed. More on that can be found at *Pyidp3 features*.

```
--- idpobjects.py        (original)
+++ idpobjects.py        (refactored)
@@ -55,7 +55,7 @@

     def product_of_types(self):
         import itertools
-        types = map(lambda x: getattr(self.idp,x), self.typing)
+        types = [getattr(self.idp,x) for x in self.typing]
         return itertools.product(*types)

 class IDPEnumeratedObject(IDPVocabularyObject):
@@ -96,7 +96,7 @@

     def product_of_types(self):
         import itertools
-        types = map(lambda x: getattr(self.idp,x), self.typing)
+        types = [getattr(self.idp,x) for x in self.typing]
         return itertools.product(*types)

     @property
@@ -162,7 +162,7 @@
         IDPGeneratedObject.__init__(self,idp,name,args,impl)

     def __getitem__(self, key):
```

(continues on next page)

(continued from previous page)

```
-         args = map(self.idp.object_for_name, key)
+         args = list(map(self.idp.object_for_name, key))
          try:
              res = self.implementation(args)
          except AttributeError:
@@ -275,4 +275,4 @@
          self.rules = rule_list

      def in_theory(self):
-         return "{\n" + "\n".join(map(lambda x: x.in_theory(), self.rules)) + "\n}"
+         return "{\n" + "\n".join([x.in_theory() for x in self.rules]) + "\n}"
--- idp_parse_out.py        (original)
+++ idp_parse_out.py        (refactored)
@@ -126,7 +126,7 @@
 def parse_tuple(tup):
      tup = tup.strip()
      elements = tup.split(',')
-     parsed = map(parse_element, elements)
+     parsed = list(map(parse_element, elements))
      if len(parsed) == 1:
          return parsed[0]
      return tuple(parsed)
@@ -144,7 +144,7 @@
 def parse_enumeration(s):
      s = s.strip()
      elements = s.split(';')
-     parsed = map(parse_enumerated, elements)
+     parsed = list(map(parse_enumerated, elements))
      if "->" in s: # Function
          return dict(parsed)
      else: # Predicate
@@ -153,7 +153,7 @@
 def parse_range(s):
      s = s.strip()
      low, up = s.split('..')
-     return range(int(low),int(up))
+     return list(range(int(low),int(up)))

 def parse_contents(s):
      stripped = s.strip().lstrip('{').rstrip('}').strip()
--- idp_py_syntax.py        (original)
+++ idp_py_syntax.py        (refactored)
@@ -44,7 +44,7 @@
          self.formula = formula

      def __str__(self):
-         var_tuple =  flatten(map(lambda x: x[0], self.vars))
+         var_tuple =  flatten([x[0] for x in self.vars])
          it = (" ".join(var_tuple) + ": " +
                  " & ".join(map(tuple_to_atom, self.vars)))
          if self.agg == "card":
@@ -63,10 +63,10 @@
              symbols =  "+-*%/^"
              return "(" + x + ")" if any([(s in x) for s in symbols]) else x
          if self.symbol == "/":
-             (l,r) = map(lambda x: add_pars(str(x)), self.children)
+             (l,r) = [add_pars(str(x)) for x in self.children]
```

(continues on next page)

(continued from previous page)

```
             return "(" + l + "-" + l + "%" + r + ") / " + r
         else:
-            return (" " + self.symbol + " ").join(map(lambda x: add_pars(str(x)),␣
↪self.children))
+            return (" " + self.symbol + " ").join([add_pars(str(x)) for x in self.
↪children])

 class BooleanFormula(Formula):

@@ -75,7 +75,7 @@
         self.children = children

     def __str__(self):
-        return (" " + self.symbol + " ").join(map(lambda x: "(" + str(x) + ")", self.
↪children))
+        return (" " + self.symbol + " ").join(["(" + str(x) + ")" for x in self.
↪children])

 class UnaryFormula(Formula):

@@ -100,7 +100,7 @@
             return "&"

     def __str__(self):
-        var_tuple =  flatten(map(lambda x: x[0], self.vars))
+        var_tuple =  flatten([x[0] for x in self.vars])
         return (self.kind + " " + " ".join(var_tuple) + ": " +
                 " & ".join(map(tuple_to_atom, self.vars)) +
                 self.guard_sym() + " "+ str(self.formula))
@@ -179,7 +179,7 @@
         return UnaryFormula(symb, self.visit(node.operand))

     def visit_Tuple(self, node):
-        return tuple(map(lambda x: self.visit(x), node.elts))
+        return tuple([self.visit(x) for x in node.elts])

     def visit_GeneratorExp(self,node):
         return self.visit_ListComp(node)
@@ -199,7 +199,7 @@
             symb = "?" if func == "any" else "!"
             return QuantifiedFormula(symb, *self.visit(node.args[0]))
         aggregates = { 'sum' : 'sum', 'len' : 'card', 'product' : 'prod', 'max' :
↪'max', 'min' : 'min' }
-        if func in aggregates.keys():
+        if func in list(aggregates.keys()):
             return AggregateFormula(aggregates[func], *self.visit(node.args[0]))
         return str(func) + "(" + ", ".join(map(str, [self.visit(arg) for arg in␣
↪node.args])) + ")"

--- test.py         (original)
+++ test.py         (refactored)
@@ -1,11 +1,11 @@
 #!/usr/bin/python3
-from typedIDP import *
+from .typedIDP import *
```

(continues on next page)

(continued from previous page)

```
 idp = IDP('/home/saltfactory/Documents/Masterproef/IDP/idp3-3.7.1-Linux/usr/local/
↪bin/idp')

-idp.Type("Student", range(int(5)))
-idp.Type("Groepnummer",range(int(1)))
+idp.Type("Student", list(range(int(5))))
+idp.Type("Groepnummer",list(range(int(1))))
 idp.Function("InGroep(Student): Groepnummer")
 idp.Constraint("""!groep[Groepnummer]: #{student[Student]:
                         InGroep(student) = groep} =< """ + str(10),True)
--- typedIDP.py        (original)
+++ typedIDP.py        (refactored)
@@ -1,8 +1,9 @@
 IDP_LOCATION = "/home/saltfactory/Documents/Masterproef/IDP/idp3-3.7.1-Linux/usr/
↪local/bin/idp"

-from idp_py_syntax import parse_formula
-
-from idpobjects import *
+from .idp_py_syntax import parse_formula
+
+from .idpobjects import *
+from functools import reduce

 class Block(object):

@@ -56,7 +57,7 @@
         return "vocabulary " + self.name

 def subclasses(cls):
-    return reduce(lambda x,y: x + y, map(lambda x: subclasses(x) + [x], cls.__
↪subclasses__()), [])
+    return reduce(lambda x,y: x + y, [subclasses(x) + [x] for x in cls.__subclasses__
↪()], [])

 class IDP(object):

@@ -103,7 +104,7 @@
     #if the second argument 'enumeration' is given, then the Type is an int
     #else, it's left blank
     def Type(self, name, enumeration):
-        if len(enumeration) > 0 and all([isinstance(x, (int,int)) for x in␣
↪enumeration]):
+        if len(enumeration) > 0 and all([isinstance(x, int) for x in enumeration]):
             res = IDPIntType(self, name, enumeration)
         else:
             res = IDPType(self, name, enumeration)
@@ -166,7 +167,7 @@
         return str(thing)

     def assign_name(self, object_):
-        if isinstance(object_, (int,int,str,bool,float)): # Primitive type
+        if isinstance(object_, (int,str,bool,float)): # Primitive type
             return object_
         name = "o" + str(id(object_))
         self.object_names[name] = object_
@@ -222,7 +223,7 @@
```

(continues on next page)

(continued from previous page)

```
         self.know(old_class(old.typedName, new))

     def __str__(self):
-        return "\n".join(map(lambda bl: bl.show(self.idpobjects.values()), self.
↪blocks)) + "\n" + IDP.gen_models + "\n"
+        return "\n".join([bl.show(list(self.idpobjects.values())) for bl in self.
↪blocks]) + "\n" + IDP.gen_models + "\n"

     def fillIn(self, pred):
         if self.dirty:
@@ -272,13 +273,13 @@
             print("GOT OUTPUT:")
             print(out)
             print("END OF IDP OUTPUT")
-        import idp_parse_out
+        from . import idp_parse_out
         if out.strip() == "nil":
             print("UNSATISFIABLE!")
             self.cache = {'satisfiable' : []}
         else:
 #            self.Predicate("satisfiable", [()])
-            self.cache = idp_parse_out.idp_parse(out, map(lambda x: x.name, self.
↪wanted))
+            self.cache = idp_parse_out.idp_parse(out, [x.name for x in self.wanted])
         self.dirty = False

     def checkSat(self):
@@ -297,8 +298,8 @@
         out,err = idp.communicate(input=str(script))

         if __debug__:
-            print("err:" + err)
-            print("out:" + out)
+            print(("err:" + err))
+            print(("out:" + out))
         #check wether the output is true or false
         if out.find("true") != -1:
             return True
@@ -343,7 +344,7 @@
 def type(idp):
     def foo(cls):
         clsname = cls.__name__
-        for name, method in cls.__dict__.iteritems():
+        for name, method in cls.__dict__.items():
             if hasattr(method, "_idp_return_type"):
                 rt = getattr(method, "_idp_return_type")
                 if isinstance(rt, str):
```

CHAPTER 5

The Pyidp3 API reference

This page contains the API reference for Pyidp3. This is an entirely autogenerated page, based on the docstring that are inside the code. Some things might be wrong, or might be outdated. If you find any of such things, let me know and I'll fix them (or make a pullrequest).

As the entire layout can be a bit awkward, I also autogenerated UML diagrams. This is the package structure:

And this is the entire class structure, with only the classnames:

For an overview of the entire class structure with methods and attributes, check the bottom of this page.

5.1 The typedIDP submodule:

This is the main, toplevel submodule and contains everything a normal user would need.

class pyidp3.typedIDP.**Block**(*name*)

An abstract superclass for the editable blocks. Should never be explicitly instantiated, but there's no elegant way to enforce abstraction in python3. These editable blocks consist of: * Theory; * Structure; * Vocabulary; * Term.

There is no Main block, see further.

Raises **NotImplementedError** – some of the methods need to be implemented by the subclass.

begin()
Generates the beginning of a Blockstring. This is just an opening bracket and enter for all blocktypes.

Returns the begin of a block in IDP-form.

Return type str

content(*objects*)
Creates the actual content of the block, based on what it contains.

Parameters **objects** (*TODO*) – TODO

Returns the content of the objects in IDP-form.

Return type str

end()
The ending of a Blockstring. This is the same for all the blocktypes. Consists of a closing bracket and a couple of enters.

Returns the ending of a block in IDP-form.

Return type str

header()
The default header raises an error if it isn't overwritten.

Raises **NotImplementedError** – if it's not overwritten.

method()

Returns todo

Return type str

show(*objects=None*)
Function to fully generate a Block in IDP-interpretable string. Every block consists of a header, a begin section, a section containing objects, and an end.

The only Block without an 'objects' variable is the Term block.

Parameters **objects** (*TODO*) – TODO

Returns the block in IDP-form.

Return type str

class pyidp3.typedIDP.**IDP**(*executable='~/idp/usr/local/bin/idp'*)
A class containing everything needed to 'use' the IDP system. It allows adding and removing new constraints, functions, relations, ... which it can then convert into a usable IDP script. This script can be piped into the idp executable, whose path is supplied in the init. The output can then be decoded and turned back into Pythonic data structures. This allows for a full interface in Python, and theoretically no much knowledge of IDP is needed. Although, it is possible (and in my opinion preferred) to supply most of the idpobjects already in IDP format, which requires IDP knowledges but removes the danger of converting Python to IDP.

Constant(*typed_name*, *enumeration=None*)
Adds a constant.

Parameters

- **typed_name** (*str*) – the name of the constant, in IDP format.
- **enumeration** – this is currently not used. TODO: FIX!

- **enumeration** – list of values

Returns the constant itself

Return type Function

Constraint(*formula*, *already_IDP=False*)

Adds a constraint. There are two types of constraint:

1. Constraints with formula in the Python form;
2. Constraints with formula in the IDP form.

In the second case, the formula doesn't need to be parsed into the IDP form. This is a 'safer' way of programming, but it requires more knowledge of the IDP system. In the first case, the Python form needs to be parsed into the IDP form. This is done by passing it on to the parse_formula function.

Parameters

- **formula** (*str*) – the formula in either Python or IDP form
- **already_IDP** (*bool*) – a bool to flag what form the formula is in

Returns an IDP object

Return type *IDPConstraint*

Define(**args*)

Method to make a definition. It can be called as "Define(Head,Body)" for a definition with only one rule, or it could be called as "Define([(H1,B1), (H2,B2), ...])" for definitions with multiple rules. As last argument, a "already_idp" flag could be passed. This function is a bit experimental, best to format it as IDP and use the 'already_idp' flag.

Parameters

- **head** (*str*) – the head of the rule
- **body** (*str*) – the body of the rule
- **already_idp** – flag of whether it's in the correct form or not

type already_idp: bool

OR

Parameters **list** (*list of tuples*) – tuples of heads and bodies

Returns the definition in IDP form

Return type *IDPDefinition*

Function(*typed_name*, *enumeration=None*, *partial=False*)

Adds a function. This function is either:

- Empty;
- Completely filled;
- Partial (not advised).

Parameters

- **typed_name** (*str*) – the name of the function, in IDP format
- **enumeration** (*dictionary*) – dictionary containing the values of the function
- **partial** – flag to make partial function (not advised)

Partial bool

Returns an IDPEnumeratedFunction or IDPUnkownFunction object

Example:

Function("Weight(Penalty): Number", [penalty1:5, penalty2:15, penalty3:30])

would be formatted to:

Weight(Penalty) = {penalty1->5, penalty2->15, penalty3->30}

GeneratedFunction(*typed_name*, *impl*)
TODO: Document this! >:(

GeneratedPartialFunction(*typed_name*, *impl*)
TODO: Document this! >:(

Predicate(*typed_name*, *enumeration=None*, *ct=False*)
Adds a predicate. It can either be empty, or already (partially) filled.

Parameters

- **typed_name** (*str*) – name of the predicate in IDP format
- **enumeration** – an x-dimensional array containing the data.

x needs to equal to the amount of variables there are in the IDP formatted predicate :type enumeration: list of str, int, float, ... :returns: the Predicate in a datastructure :rtype: IDPEnumeratedPredicate or IDPUnknownPredicate

Example:: Predicate("IconicDuo(Character,Character)", [["Harry","Sally"], ["Bonny","Clyde"]]

would result in:: "IconicDuo(Charachter,Character) = {(Harry,Sally); (Bonny,Clyde)}"

Type(*name*, *enumeration*, *constructed_from=False*, *isa=None*)
Adds a type. The type can be int or stringbased. The enumeration needs to be supplied as a list, or a tuple. As for right now, string should be supplied with extra quotation marks.

The int can be supplied as a list of ints or as a tuple of ints. A tuple can be used to set a range of values, and will be translated as such.

Example Type(Example, (0,10)) -> Example = {0..10} Type(Example, list(range(0,10))) -> Example = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

Parameters

- **name** – the name of the Type
- **enumeration** – a one-dimensional list containing all possible

values of the Type. :param constructed_from: allows the Type to have 'constructed_from'. TODO: Add this! :type name: str :type enumeration: list of int, string, float, ... :type constructed_from: list of int, string, float, ... :returns: IDPIntType or IDPType object

append(*p*)

Adds objects into the idpobjecst dictionary. These objects are:

- IDPPredicate, IDPUnknownPredicate
- IDPType, IDPInt
- IDPUnknownFunction, IDPEnumeratedFunction
- Function

- IDPGeneratedFunction
- IDPGeneratedPartialFunction
- IDPDefinition

Parameters **p** (*see above*) – one of the above listed objects

check_sat()
Checks the satisfiability of the current IDP system.

check_sat_script()
Generates .idp file to check if the model is satisfiable.

Returns A script readable by IDP.

Return type str

customScript(*main*, *term=''*)
Generates a .idp file with a custom main. Works by adding a custom Mainblock to the other blocks. Optionally, a term to minimize can also be added. The Term needs to be preformatted Term string.

Returns A custom script readable by IDP.

Return type str

Raises ValueError when the supplied term isn't str

forget(*old*)
Delete an object from the idpobjects dictionary, and also from the wanted list if it was found there.

Parameters **old** (*an idpobject*) – the idpobject to remove

init_options()
This method initialises all the options. They all start as None-values (safe for xsb and nbrmodels)

know(*p*)
Adds objects into the idpobject dictionary, and returns the object. These objects are:

- IDPPredicate, IDPUnknownPredicate
- IDPType, IDPInt
- IDPUnknownFunction, IDPEnumeratedFunction
- Function
- IDPGeneratedFunction
- IDPGeneratedPartialFunction
- IDPDefinition

Parameters **p** (*see above*) – one of the above listed objects

Returns the object which was added to the dictionary

minimize(*term*, *ssh=False*, *remote_idp_location=None*, *known_hosts_location=None*, *address=None*, *username=None*, *password=None*)
Run the IDPsystem's minimize and read its output. Works by piping to input to the IDP executable, and reading the output. Once this command has been run, the idp object should have new attributes in the same name of the constants/functions/relations/..., which should be readable in Python.

For example, a function called 'Group' should now be accessible by fetching the 'Group' attribute of the IDP object.

Parameters

- **term** (*bool*) – the content of the term block, in IDP-form
- **ssh** – Can be used to run IDP over SSH

minimize_script(*termblock*)

Generates the script for basic minimization of a term. Works by adding a Term and a Main block to the other blocks. More specifically: a Main block containing the 'minimize' function. The Term block needs to be called 't' and needs to use Vocabulary 'V' for it to work. By default a voc is always called V, so this is no problem.

Returns A script readable by IDP for term minimization.

Return type str

modelexpand_script()

Generates the IDP-ready script for basic modelgeneration. Works by adding a Main block to all the other blocks. More specifically: a Main block containing the 'generate' IDP function.

Returns A fullfledged IDP-readable script for modelgeneration.

Return type str

printunsatcore(*timeout=0*)

Call printunsatcore on the IDP object and return the unsat core. The call will end early after *timeout* seconds. (default: 0 = no timeout)

refresh()

Run the IDPsystem's modelgeneration and read its output. Works by piping to input to the IDP executable, and reading the output. Once this command has been run, the idp object should have new attributes in the same name of the constants/functions/relations/..., which should be readable in Python.

For example, a function called 'Group' should now be accessible by fetching the 'Group' attribute of the IDP object. :Example:

grouparray = IDP.Group

static split_func_name(*func_name*)

Static method to split a function. Splits a Function name into two parts: the function, and the Type it maps on (the return_type).

Example Foo(bar): baz would be split in "Foo(bar)" and "baz".

static split_pred_name(*pred_name*)

Static method to split a predicate. Splits a predicate in two parts: the name of the predicate, and the type it holds.

Example Foo(bar,baz) would be split in "Foo" and "[bar, baz]"

class pyidp3.typedIDP.**Structure**(*name*, *voc*)

A class for the Structure block, which is a subclass of Block. It adds the voc-attribute, and overwrites the header.

Inherits Block

header()

Generates the specific header for a Structure.

Returns the header, in IDP-form.

Return type str

class pyidp3.typedIDP.**Term**(*term*, *voc='V'*)

A class for the Term block, which is a subclass of Block. It adds a term attribute, (a string containing the content of the Term block) and overwrites the header and the content methods.

content()
Term has a specific content, which is just the self.term (cause it's already in IDP format). This is between two linefeeds.

Returns the termcontent as it was initialized (self.term)

Return type str

header()
The specific header for a Term.

Returns the Termblock turned into IDP format

Return type str

class pyidp3.typedIDP.**Theory**(*name*, *voc*)
A class for the Theory block, which is a subclass of Block. It changes the voc-attribute, and overwrites the header.

Inherits Block

header()
The specific header for a Theory, the only variables are the Theoryname and the Vocabularyname.

For Theory T and Vocabulary V, the string looks like: 'theory T: V '

Returns the header of the Theory, in IDP-form.

Return type str

class pyidp3.typedIDP.**Vocabulary**(*name*)
A class for the Vocabulary block, which is a subclass of Block. It overwrites the header. Uses Block's __init__ method.

header()
Generates the specific header for a Vocabulary.

Returns the header, in IDP-form.

Return type str

pyidp3.typedIDP.**subclasses**(*cls*)
TODO: describe

5.2 The idpobjects submodule:

This submodule contains an object for every kind of idp object.

This file contains all the IDP objects.

class pyidp3.idpobjects.**IDPConstraint**(*idp*, *formula*)

class pyidp3.idpobjects.**IDPConstructedType**(*idp*, *name*, *enum*, *ct=False*)

class pyidp3.idpobjects.**IDPDefinition**(*idp*, *rule_list*)

class pyidp3.idpobjects.**IDPEmptyConstantFunction**(*idp*, *name*, *args*, *rt*, *partial=False*)

class pyidp3.idpobjects.**IDPEnumeratedFunction**(*idp*, *name*, *args*, *rt*, *enum*, *partial=False*, *ct=False*)

class pyidp3.idpobjects.**IDPEnumeratedObject**(*idp*, *name*, *typing*, *enum*, *ct*)

class pyidp3.idpobjects.**IDPEnumeratedPredicate**(*idp*, *name*, *typing*, *enum*, *ct*)

add(*x*)
Add an element.

discard(*x*)
Remove an element. Do not raise an exception if absent.

class pyidp3.idpobjects.**IDPFloatRangeType**(*idp*, *name*, *enum*, *ct=False*)

class pyidp3.idpobjects.**IDPFloatType**(*idp*, *name*, *enum*, *ct=False*)

class pyidp3.idpobjects.**IDPFunction**(*idp*, *name*, *types*, *return_type*, *partial=False*, *ct=False*)

class pyidp3.idpobjects.**IDPGeneratedFunction**(*idp*, *name*, *args*, *rt*, *impl*, *partial=False*)

class pyidp3.idpobjects.**IDPGeneratedObject**(**args*)

class pyidp3.idpobjects.**IDPGeneratedPredicate**(**args*)

class pyidp3.idpobjects.**IDPIntRangeType**(*idp*, *name*, *enum*, *ct=False*)

class pyidp3.idpobjects.**IDPIntType**(*idp*, *name*, *enum*, *ct=False*)

class pyidp3.idpobjects.**IDPObject**(*idp*)
'Abstract' class for all the IDP objects. Initialises idp.

class pyidp3.idpobjects.**IDPPredicate**(*idp*, *name*, *typing*, *ct=False*)

class pyidp3.idpobjects.**IDPRule**(*idp*, *head_pred*, *vars_*, *body*)

class pyidp3.idpobjects.**IDPRuleStr**(*idp*, *string*)

class pyidp3.idpobjects.**IDPSpecialType**(*idp*, *name*, *enum*, *ct=False*)

class pyidp3.idpobjects.**IDPTheoryObject**(*idp*)

class pyidp3.idpobjects.**IDPType**(*idp*, *name*, *enum*, *ct=False*)

class pyidp3.idpobjects.**IDPUnknownFunction**(*idp*, *name*, *args*, *rt*, *partial=False*, *ct=False*)

class pyidp3.idpobjects.**IDPUnknownObject**(*idp*, *name*, *typing*, *enum*, *ct*)

class pyidp3.idpobjects.**IDPUnknownPredicate**(*idp*, *name*, *types*)

class pyidp3.idpobjects.**IDPValueConstantFunction**(*idp*, *name*, *args*, *rt*, *enumeration*, *partial=False*)

class pyidp3.idpobjects.**IDPVocabularyObject**(*idp*, *name*, *typing*, *ct=False*)
Abstract class for all the objects that appear in a vocabulary.

in_theory()
Doesn't show up in a theory, so returns empty string.

5.3 The idp_py_syntax submodule:

This submodule contains everything needed to convert Pythonic data to IDP. This is something that was support by the original Pyidp, but is no longer supported by Pyidp3. The code might still work, it's just not being worked on.

class pyidp3.idp_py_syntax.**FormulaBuilder**

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

5.4 The idp_parse_out submodule:

This submodule contains all the code necessary to read IDP output and convert it to Python.

Python Module Index

p

Index

K

M

P

R

S

T

V