БИБЛИОТЕКА DECPY

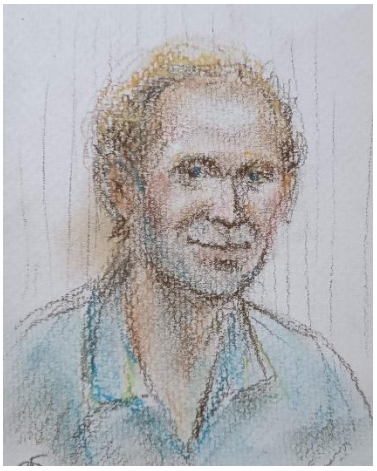
Декларативное программирование на Python

П.В.Добряк

УДК 004.432

ББК 32.973.25

Учебник по применению библиотеки DecPy, которая встраивает в язык Python ленивые вычисления и три вида декларативного программирования: исчисление на кортежах (аналог языка SQL), исчисление на доменах (аналог QBE) и исчисление предикатов первого порядка (аналог Prolog, логическое программирование), позволяя работать с коллекциями в Python как с базами данных или экспертными системами (в том числе писать рекурсивные запросы и даже доказывать теоремы).



Об авторе: Павел Вадимович Добряк – кандидат технических наук, преподаватель Уральского федерального университета, ведущий занятия по различным языкам программирования, базам данных, искусственному интеллекту и проектированию информационных систем. Автор книг про программированию на Python. Репетитор математики и информатики. Область научных интересов: сложные модели данных и алгоритмы, мультипарадигменное программирование. **Создатель библиотеки DecPy.**

Добряк П.В. Библиотека DecPy. Декларативное программирование на Python. – Электронное издание. Екатеринбург, 2025. – 144 с.: ил.

Содержание

Введение

«Рекламный обзор» - что такое библиотека DecPy

Когда я закончил разработку библиотеки DecPy, то у меня возникли сложности с тем, как объяснить различным людям, что именно я сделал; тем более, что жизнь все ускоряется, и объяснения часто приходится делать на бегу. В итоге я выработал короткие ответы, характеризующие библиотеку DecPy. Приведу эти ответы в рекламно-презентационном стиле:

Для начинающих программистов школьного уровня:

Во многих случаях вы сможете с помощью библиотеки DecPy вместо десятка строк запутанного кода написать код в одну понятную строчку.

Для программистов, знающих также базы данных:

Библиотека DecPy позволяет писать запросы к коллекциям Python и классам, как будто это таблицы из базы данных. Причем она встраивает не только возможности языка SQL, но и альтернативы SQL – языка QBE, а также языка искусственного интеллекта Prolog (с помощью чего легко решаются, например, задачи по выборкам из графов, которые на стандартном SQL делать затруднительно). Библиотека позволяет комбинировать эти средства в одном запросе.

Для программистов, знающих C# и LINQ:

Библиотека DecPy – это фактически LINQ для Python, но только с более богатыми возможностями. В частности, в Python библиотека встраивает не только аналог SQL, но и QBE и Prolog. Возможно комбинирование этих средств в одном запросе, а также рекурсивные запросы.

Для программистов с университетским образованием:

Библиотека DecPy расширяет мультипарадигменный язык Python комбинацией трех видов декларативного программирования, в основе которых лежит исчисление на кортежах (аналог SQL), исчисление на доменах (аналог QBE), исчисление предикатов первого порядка (аналог Prolog, логическая парадигма программирования). При этом используются стандартные операторы Python и минимальный набор функций-декораторов. Таким образом, DecPy – это не просто очередная библиотека с полезным набором функций. Ее средства органически встраиваются в Python – речь идет о расширении и следующем шаге в развитии самого языка Python.

Предлагаю вашему вниманию небольшую книгу – учебник по библиотеке DecPy.

Если вы являетесь таким профессионалом, что поняли все «рекламные» абзацы для разных категорий программистов, то, возможно, вместо 120-страничной книги вам проще будет прочитать 15-страничный обзор библиотеки DecPy в приложении. Так же стоит поступить и программистам с большим практическим опытом, позволяющим интуитивно понимать код.

Для полных новичков, возможно, стоит пропустить Предисловие и Исторический обзор и сразу приступить к изучению первого раздела, а потом вернуться к Введению после прочтения второго раздела.

Предисловие

Python является языком высокого уровня и общего назначения. На нем можно решать широкий круг задач, а удобные библиотеки функций позволяют разрабатывать приложения практически для всех отраслей ИТ-индустрии. Python обладает богатым набором языковых конструкций, позволяющих писать программы в разных стилях (парадигмах программирования). Являясь мультипарадигменным языком, Python позволяет писать программы в структурном, процедурном, функциональном и объектно-ориентированном стиле.

Вместе с тем, если писать на чистом Python запросы-выборки из баз данных, задания экспертным системам, решать задачи поиска оптимального пути, то это потребует от программиста хорошего алгоритмического мышления и написания десятка строк кода. Но есть специализированные системы со своими языковыми средствами, в которых эти задачи легко решаются людьми, которые даже могут не быть программистами. Эти языковые средства можно назвать ***декларативным программированием***.

В учебнике описывается библиотека DecPy (разработанная автором книги), встраивающая в язык Python декларативное программирование нескольких видов. Фактически библиотека расширяет язык Python. С помощью нескольких ключевых слов уже известные языковые конструкции (например, квадратные скобки) изменяют свою работу.

В результате с помощью библиотеки DecPy для сложных задач можно писать понятный и краткий код в одну строчку, заменяющий десятки строк кода на стандартном Python.

Для изучения книги читатель должен знать язык Python на уровне основных языковых средств: циклов, условий, функций, списков и множеств, в небольшом объеме – классов (на уровне «что такое класс и его экземпляр, атрибут и метод»). Знание декораторов и конструкции lambda желательно, но не обязательно. Также желательно, но не обязательно знать оператор select из языка SQL (его знание позволит оценить красоту предлагаемых решений библиотеки DecPy).

Терминология и исторический обзор

«Декларативное программирование на Python! А это вообще возможно?» - скажет читатель, знакомый с декларативным программированием, если он писал запросы к базам данных или, получая хорошее образование в университете, писал экспертные системы на довольно уже древнем языке искусственного интеллекта Prolog (до той поры, пока почти все задачи, связанные с искусственным интеллектом, не начали решать с помощью нейронных сетей).

Для читателей, незнакомых с термином ***декларативное программирование***, поясню, что есть два крупных стиля программирования: императивное и декларативное. В императивном программировании программист пишет алгоритм на языке программирования в виде последовательности шагов. В алгоритме могут быть условия, в зависимости от которых программа будет выполняться по одной или другой ветке, а также циклы (повторения шагов).

«Получается, я программирую в императивном стиле», - скажет читатель, и будет совершенно прав. Обычно программированию сейчас учатся на одном из языков высокого уровня, например, Python или языках семейства C++: C++, java, C#, теперь к этому списку добавился Kotlin. А раньше учились на Fortran, Basic и Pascal.

Императивный стиль – это обобщенное название для нескольких стилей (парадигм) программирования. Если вы используете только вложенные друг в друга циклы и условия, вы программируете в ***структурном стиле***. Если куски программ (маленькие отлаженные и законченные по смыслу алгоритмы, составляющие большую программу) помещаете в функции (процедуры), то это – ***процедурный стиль***. Если же вы данные объединяете с кодом, который их обрабатывает, описывая предметную область в виде системы классов, то вы программируете в ***объектно-ориентированном стиле***.

Небольшая часть айтишников начинала с баз данных. И именно при работе с базами данных происходит знакомство с декларативным программированием, когда делаются выборки из баз данных.

«Выборки из баз данных – разве это программирование? - спросит читатель. - Я не пишу алгоритм, я просто описываю, что я хочу получить». Например, если таблица содержит торговую информацию о сделках, я пишу: «Выведи продавцов, у которых суммарная стоимость сделок за неделю была больше 100 000 руб».

«Вот вы и описали, что такое декларативное программирование!» – отвечу я. В ***декларативном программировании*** мы не пишем алгоритм получения результата, а даем задание в виде формулировки результата, который хотим получить. Алгоритм получения результата система управления базой данных сочиняет за нас сама.

Действительно, языки запросов к базам данных задумывались не как языки программирования, а как нечто очень простое, позволяющее даже не программисту сразу начать писать запросы для выборок из таблиц. Конечно, задания бывают очень сложными и их надо еще научиться формулировать. Таков язык программирования ***SQL***, который изначально задумывался как SEQUEL – Structure English Query Language – язык запросов на английском языке. Предполагалось, что пользователь будет писать запросы на своем родном английском языке, пусть чуть-чуть более формализованном, чем живой разговорный. Язык SQL породил множество диалектов для различных типов баз данных.

А еще в базах данных есть мастера выборок данных, где выборка делается вообще без написания кода. Пользователь лишь перетаскивает колонки из таблиц в результат, прописывая, что надо взять в чистом виде, а что – посчитать (например, сумму). За этими мастерами, как правило, прячется язык ***QBE*** – Query By Example («запрос по образцу»). Предполагается, что пользователь базы данных конструирует образец того, что хочет получить. А СУБД сама подбирает алгоритм по этому образцу и находит решения.

Небольшая часть программистов, теперь в основном по университетскому образованию, знакома с языком искусственного интеллекта ***Prolog***. Допустим, задано родословное древо в виде очень коротких записей, представляющих собой пары (ребенок, мать) и (ребенок, отец). Как найти бабушку или дедушку? Очень просто, бабушка – это мать матери или мать отца. Формулировка понятий в виде поиска по родословному древу очень похожа на запросы к базам данных. Но есть большая смысловая разница: мы не просто

делаем запросы на выборку данных, мы формулируем новые понятия. Попробуйте вот сформулировать, кто такой предок? И язык Prolog для этого более удобен, чем SQL.

В 2010-е годы был бум развития геоинфомационных систем. Многим случалось добираться самолетами с несколькими пересадками из одного города в другой. Совершенно типичный запрос: «Выведи все способы добраться из одного города в другой без пересадок или с любым количеством пересадок и посчитай их стоимость». Эта, в общем, типовая задача, как ни странно, вызовет большие затруднения как на SQL, так и на Prolog. В 1990-е и 2000-е годы интенсивно развивались специализированные средства для работы с графами и другими средствами геоинформационных систем. Но рискну высказать свое мнение (оно может не совпасть с мнением специалистов), что какого-то общепризнанного и удобного декларативного средства для работы с графами так и не появилось. Когда вы пользуетесь сайтом для подбора самолетных рейсов, знайте, что «за ним» прячется большой интеллектуальный труд программистов с несколькими десятками строк кода.

Получается, что языки декларативной парадигмы сейчас – это специализированные языки для определенных типов предметных областей (баз данных, экспертных систем, геоинформационных систем). Когда-то так и считалось, что будущее – за декларативными языками, и что постепенно для большинства типов предметных областей будут разработаны свои специализированные декларативные языки, и что профессионалы в этих предметных областях будут ими пользоваться, не будучи программистами.

Естественно, базы данных и экспертные системы – это часть ИТ-индустрии, и потребовалась их интеграция в информационные системы (хотя бы в виде отображения на сайтах). Для языков общего назначения (высокого уровня) были разработаны библиотеки для работы с ними. Как правило, эти библиотеки содержат классы, описывающие предметную область (например, таблицы) и функции для обработки данных. Причем эти функции принимают задания (запросы) на выборку в виде текстовой строки (в большинстве случаев – запрос на языке SQL), которую затем компилируют. То есть речь идет о самой поверхностной интеграции, когда текстовые строки из кода на языке высокого уровня компилируются библиотечными функциями как язык SQL. Это можно назвать интеграцией в техническом смысле, но не в языковом.

Самое известное исключение из этого – язык ***LINQ***, встроенный в Microsoft Visual Studio. Можно писать код на SQL-подобном языке LINQ внутри

программы C#, причем этот код – органическая часть C# на уровне языковых конструкций и переменных.

Язык Python был моим следующим языком после C#. С течением времени Python мне нравился все больше. Но все же наличие LINQ в C# мне казалось большим преимуществом перед Python. Это казалось не мне одному. Многие ожидали появления аналогов LINQ и в Python, пока создатель языка Гвидо Ван Россум не пояснил, что официального LINQ в Python не будет, потому что Python обладает достаточным набором языковых конструкций, чтобы решать задачи, которые на C# решаются с помощью LINQ.

Часть разработчиков восприняла эти слова с сожалением. Но, комбинируя различные приемы функционального и объектно-ориентированного, а сейчас и аспектно-ориентированного программирования, можно самим написать LINQ-подобные системы для Python. И такие системы стали появляться. Из распространенных следует отметить SQLLight, с помощью которого на Python можно создать базу данных и писать к ней запросы на языке SQL, который органично встроился в Python.

Изучая эти приемы продвинутого программирования, стал экспериментировать с написанием LINQ-подобных систем для Python и я. Но почему надо останавливаться именно на языке SQL? Я пробовал писать прототипы QBE и Prolog. При постановке задачи для меня важно было то, что разработанная система должна быть органичной частью языка Python, а не интерпретатором строк, чтобы получилось расширение языка Python с минимумом новых ключевых слов (функций) и чтобы при решении задач использовались языковые средства самого Python, пусть немного измененные. Мне удалось сделать это с помощью 4 основных ключевых слов и 3 дополнительных, импортируемых из библиотеки.

Анализируя три вида декларативного программирования в языках SQL, QBE и Prolog, я видел их преимущества для определенного типа предметных областей и недостатки для других. И, разрабатывая прототипы LINQ для этих трех разновидностей декларативного программирования, я всё чаще думал об их объединении. Действительно, если я пользуюсь одними и теми же языковыми средствами Python, почему бы не сделать их совместное применение для SQL, QBE и Prolog?

Это объединение мне удалось сделать, и я представляю читателю библиотеку DecPy. Для задачи самолетных рейсов с пересадкой такое объединение оказалось очень удачным. Решение этой задачи по отдельности на SQL или Prolog требует более громоздкого и некрасивого кода.

Эта книга – не просто инструкция к библиотеке DecPy, а небольшой учебник по использованию библиотеки, который позволит решать сложные задачи с гораздо меньшими интеллектуальными усилиями, чем раньше.

Установка библиотеки

Скачать библиотеку DecPy и другие полезные файлы можно с репозитория библиотеки на веб-сервисе для хостинга IT-проектов github:

https://github.com/pauldobriak/DecPy

Также библиотека опубликована в каталоге программного обеспечения, написанного на языке программирования Python PyPI. Установить ее можно стандартным путем с помощью команды в Командной строке Windows, как это делается и с другими библиотеками:

```
pip install decpy
```

1. Ленивые вычисления

Перед тем, как писать запросы (делать выборки) к коллекциям Python, рассмотрим ленивые вычисления – вспомогательную технологию, которая лежит в основе запросов.

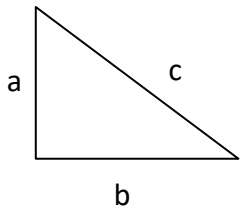
Ленивые вычисления (отложенные вычисления) – это прием программирования, появившийся в функциональной парадигме программирования. При помощи этого приема программирования данные вычисляются не в том месте, где они описаны, а там, где они реально нужны.

Ленивые вычисления не встроены в Python, они реализуются в различных библиотеках, в том числе – в библиотеке DecPy, написанной автором этой книги.

1.1. Ленивые формулы

Задача. Даны два числа – длины катетов прямоугольного треугольника. Нужно посчитать гипотенузу.

Вычисление гипотенузы происходит по теореме Пифагора:



$$c = \sqrt{a^2 + b^2}$$

Классическая программа на языке Python выглядит так:

```
a=3
b=4
c=(a**2+b**2)**0.5
print(c)
```

Результат:

```
5.0
```

В программе вместо корня используется эквивалентная ему степень ½.

Посчитаем длины гипотенуз в двух треугольниках (возьмем вторую пифагорову тройку):

```
a=3
b=4
c=(a**2+b**2)**0.5
print(c)
a=5
b=12
c=(a**2+b**2)**0.5
print(c)
```

Результат:

```
5.0
13.0
```

При множественных вычислениях по одной и той же формуле многократно повторять один и тот же код нерационально. Классическим решением является выделение формулы в функцию:

```
def pithagor(a,b):
    return (a**2+b**2)**0.5

print(pithagor(3,4))
print(pithagor(5,12))
```

Это ***процедурный стиль*** программирования, в котором повторяющиеся куски кода помещаются в функции.

Для функций, содержащих одну формулу, есть вторая форма записи через ключевое слово `lambda`:

```
pithagor=lambda a,b: (a**2+b**2)**0.5

print(pithagor(3,4))
print(pithagor(5,12))
```

Формы записи функций через `lambda` являются основой ***функционального стиля*** программирования с множеством интересных приемов. Название оператора, `lambda`, происходит из ***лямбда-исчисления*** – математической основы функционального программирования.

Вспомним, что функции в процедурном стиле программирования – это маленькие отлаженные алгоритмы, решающие фрагменты большой задачи и состоящие в общем случае из нескольких шагов, в том числе с циклами и ветвлениями. То есть функция – это мощная языковая конструкция. Можно поставить вопрос о целесообразности вынесения отдельных формул в функции («не стреляем ли мы из пушек по воробьям?»).

Альтернативой использованию функций - как в процедурном, так и в функциональном стиле - являются ***ленивые вычисления***. В стиле ленивых вычислений мы объявляем переменные и пишем вычислительную формулу. Далее мы инициализируем переменные, а формула вычисляется тогда, когда её результат становится необходимым, например, при выводе на экран.

Ленивых вычислений, встроенных в Python, нет. Воспользуемся разработанной Добряком П.В. библиотекой DecPy, из которой импортируем объявление переменной `var`:

```
from decpy import var

a=var()
b=var()
pithagor=(a**2+b**2)**0.5
a(3)
b(4)
print(pithagor)
```

Здесь строчки вида:

```
a=var()
```

объявляют, но не инициализируют переменную, а

```
a(3)
```

задают значение переменной.

При объявлении множества переменных можно воспользоваться более короткой записью. У var есть необязательный числовой аргумент – количество переменных:

```
from decpy import var

a,b = var(2)
pithagor=(a**2+b**2)**0.5
a(3)
b(4)
print(pithagor)
```

Подобный стиль программирования, вообще говоря, противоречит исходным предпосылкам создания языка Python, в котором в большинстве случаев не надо объявлять переменные, да и вообще принципам языков программирования высокого уровня, в которых переменные в начале инициализируются, а формулы вычисляются сразу же в тех местах, где они приводятся. Но ленивые вычисления – это мощный инструмент для дальнейших языковых средств, которые будут описаны в этой книге.

Пока код, содержащий ленивые вычисления, можно воспринимать как способ оформления физических задач, при котором сперва приводятся физические величины и формулы, их связывающие, а уж потом по ним делаются конкретные вычисления с числовыми значениями.

1.2. Ленивые формулы как функции

Продолжим пример из предыдущей главы и посмотрим, как ленивая формула справится с несколькими числовыми наборами переменных:

```
from decpy import var

a,b = var(2)
pithagor=(a**2+b**2)**0.5
a(3)
b(4)
print(pithagor)
a(5)
b(12)
print(pithagor)
```

Результат:

```
5.0

13.0
```

Как видно, формула обработала несколько наборов исходных данных, но повторяющийся код инициализации переменных:

```
a(3)
b(4)
```

выглядит неизящно.

Вспомним, что мы сократили код:

```
a=var()
b=var()
```

до:

```
a,b = var(2)
```

Есть ли возможность записать в одну строчку:

```
a(3)
b(4)
```

?

Библиотека DecPy позволяет инициализировать переменные прямо при вызове вычисления формулы!

```
from decpy import var

a,b = var(2)
pithagor=(a**2+b**2)**0.5
print(pithagor(3,4))
print(pithagor(5,12))
```

Здесь ленивая формула по способу своего использования превратилась в функцию.

Мы рассмотрели довольно простую формулу. Приведем более сложную формулу, в которой одна и та же переменная встречается в нескольких местах:

```
from decpy import var

a,b = var(2)
f=a**2+a*b+a+1
print(f(2,3))
```

Результат:

```
13
```

При вызове формулы как функции, числовые значения в скобках инициализируют переменные в порядке появления в формуле. В нашем примере: `a=2`, `b=3`.

Не всякая формула может быть записана с помощью ленивых вычислений библиотеки DecPy. Например, следующая формула вызовет ошибку:

```
from decpy import var

a = var()
f=2*a
print(f(3))
```

Результат:

```
TypeError: unsupported operand type(s) for *: 'int' and
'var'
```

Ошибка связана с особенностями перегрузки операторов в объектно-ориентированном программировании на Python. В большинстве случаев формулу можно адаптировать под ленивые вычисления. В нашем случае – переставить местами множители:

```
from decpy import var

a = var()
f=a*2
print(f(3))
```

Результат:

```
6
```

Что делать, если операция некоммутативна? В случае возведения в степень, если переменная – основание, то программа работает правильно:

```
from decpy import var

a = var()
f=a**2
print(f(3))
```

Результат:

```
9
```

А вот если основание – число, а показатель – переменная, то получится ошибка:

```
from decpy import var

a = var()
f=2**a
print(f(3))
```

Результат:

```
TypeError: unsupported operand type(s) for ** or pow():
'int' and 'var'
```

«Секрет» заключается в том, что левый операнд должен быть переменной. Как решить проблему, мы покажем в главе 1.4.

1.3. Сохранение «лени»

На основе одной ленивой формулы может строится другая. При этом «лень» сохраняется:

```
from decpy import var

a,b,c=var(3)
f=a+b
a(1)
b(2)
c(3)
g=c+f
print(g)
a(4)
print(g)
```

Результат:

```
6
9
```

Заметим, что вычисления в ленивых формулах выполняются тогда, когда результат реально нужен, например, при выводе на экран. Можно, однако, нарушить «лень», принудительно вычислив значение формулы. Для этого напишем после формулы круглые скобки:

```
from decpy import var

a,b,c=var(3)
f=a+b
a(1)
b(2)
c(3)
g=c+f()
print(g)
a(4)
print(g)
```

Результат:

```
6
6
```

1.4. Встройка функций в ленивые вычисления, ленивые функции

В формулу могут входить не только арифметические операции, но и функции. Например, в задаче про гипотенузу мы для извлечения корня использовали возведение в степень ½. Сделаем отдельную функцию для извлечения корня. В случае, если в функции используется формула только с арифметическими операциями, начинающаяся с переменной, ничего дополнительного делать не нужно:

```
from decpy import var

def sqrt(x):
    return x**0.5

a,b = var(2)
pithagor=sqrt(a**2+b**2)
print(pithagor(3,4))
```

Вспомним, что извлечение квадратного корня есть в библиотеке math. Прямое его использование вызовет ошибку:

```
from math import sqrt
from decpy import var

a,b = var(2)
pithagor=sqrt(a**2+b**2)
print(pithagor(3,4))
```

Результат:

```
TypeError: must be real number, not expr
```

Библиотечную функцию `sqrt` надо подготовить – преобразовать с помощью второго, помимо `var`, инструмента библиотеки DecPy – функции `lazyfun`:

```
from math import sqrt
from decpy import var,lazyfun

sqrt=lazyfun(sqrt)

a,b = var(2)
pithagor=sqrt(a**2+b**2)
print(pithagor(3,4))
```

Аналогично подготавливаются к использованию другие функции библиотеки math, например, тригонометрические.

Также в подобной подготовке нуждаются функции, написанные в самой программе, если они используют библиотечные функции math. Напишем собственную функцию вычисления корня, в которую «спрячем» вызов библиотечной функции. `Lazyfun` здесь будет применяться как декоратор:

```
import math
from decpy import var, lazyfun

@lazyfun
def sqrt(x):
    return math.sqrt(x)

a,b = var(2)
pithagor=sqrt(a**2+b**2)
print(pithagor(3,4))
```

Вернемся к проблеме, описанной в главе 1.2, когда левый операнд в формуле – число, а правый - переменная:

```
from decpy import var

x = var()
f = x+2**x
print(f(3))
```

Результат

```
TypeError: unsupported operand type(s) for ** or pow():
'int' and 'var'
```

Проблема решается вынесением `2**x` в функцию:

```
from decpy import var,lazyfun

@lazyfun
def exp(n):
    return 2**n

x = var()
f = x + exp(x)
print(f(3))
```

Результат:

```
11
```

Если вы не хотите создавать отдельную функцию, можно воспользоваться конструкцией `lambda`:

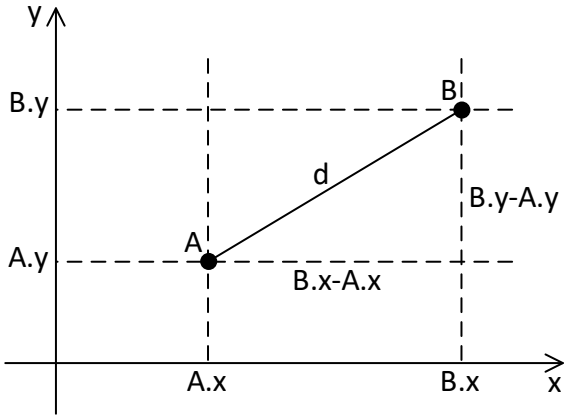
```
from decpy import var,lazyfun
x = var()
f = x + lazyfun(lambda x: 2**x)(x)
print(f(3))
```

1.5. Ленивые атрибуты

До сих пор в формулах мы использовали числа. Но в формулах могут быть более сложные объекты со своими атрибутами. Например, решим следующую задачу:

Задача. На плоскости заданы координатами две точки в декартовой системе координат. Найти расстояние между ними.

Эта задача похожа на предыдущую и также решается по теореме Пифагора:



$$d = \sqrt{(B.x - A.x)^2 + (B.y - A.y)^2}$$

Но с программной точки зрения её лучше решать в объектно-ориентированном стиле. Сделаем класс «Точка» и функцию вычисления расстояния между двумя точками. Классическое решение:

```
class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y

def dist(A,B):
    return ((A.x-B.x)**2+(A.y-B.y)**2)**0.5

A=point(0,0)
B=point(3,4)
print(dist(A,B))
C=point(5,12)
print(dist(A,C))
```

Результат:

```
5.0
13.0
```

Вместо функции вычисления расстояния напишем ленивую формулу:

```
from decpy import var

class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y

A,B=var(2)
dist=((A.x-B.x)**2+(A.y-B.y)**2)**0.5

p1=point(0,0)
p2=point(3,4)
print(dist(p1,p2))
p3=point(5,12)
print(dist(p1,p3))
```

В программе ленивая формула `dist` используется как функция. Если же вкладывать смысл в `dist` как расстояние между конкретными точками `A` и `B`, то тогда возможно такое её использование:

```
from decpy import var

class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y

A,B=var(2)
dist=((A.x-B.x)**2+(A.y-B.y)**2)**0.5

A(point(0,0))
B(point(3,4))
print(dist)
B(point(5,12))
print(dist)
```

Можно возразить, что точку `B` мы фактически подменяем другой точкой с координатами (5,12), поэтому изменим во втором случае координаты `x` и `y` точки `B` по отдельности:

```
from decpy import var

class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
```

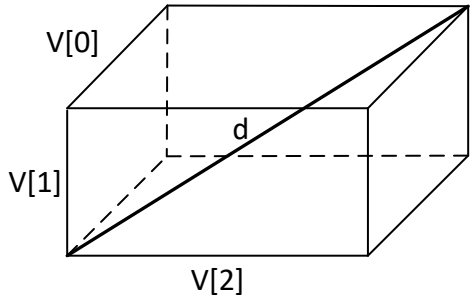
```
A,B=var(2)
dist=((A.x-B.x)**2+(A.y-B.y)**2)**0.5

A(point(0,0))
p=point(3,4)
B(p)
print(dist)
p.x=5
p.y=12
print(dist)
```

Программа отлично работает! Заметим, что когда мы написали формулу расстояния между точками, программа еще не знала класс переменных `A` и `B` и их атрибуты. Здесь мы имеем дело с таким мощным средством библиотеки DecPy, как ***ленивые атрибуты***.

1.6. Ленивые индексы

Подобно ленивым атрибутам, у переменных типа `var` есть и ***ленивые индексы***. Перенесем теорему Пифагора в пространство.

Задача. Пусть прямоугольный параллелепипед задан длинами трех сторон. Найти длину диагонали параллелепипеда. Не будем вводить отдельные переменные для трех сторон. Зададим три стороны списком.



$$d = \sqrt{v[0]^2 + v[1]^2 + v[2]^2}$$

Программа имеет вид:

```
from decpy import var

v=var()
mod=(v[0]**2+v[1]**2+v[2]**2)**0.5
print(mod([2,3,6]))
print(mod([1,4,8]))
```

Результат:

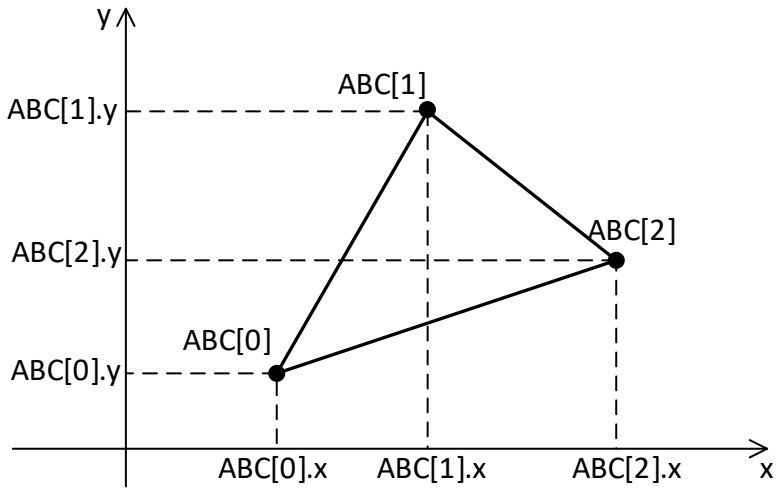
```
7.0
9.0
```

1.7. Совместное использование ленивых индексов и атрибутов

Ленивые индексы и атрибуты можно использовать совместно.

Задача. Треугольник задан координатами вершин. Найти его периметр.

Точки сделаем экземплярами класса `point`, а сам треугольник зададим списком его вершин:



Программа имеет вид:

```
from decpy import var

class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y

ABC = var()
perimeter = (((ABC[0].x-ABC[1].x)**2+(ABC[0].y-
ABC[1].y)**2)**0.5+((ABC[1].x-ABC[2].x)**2+(ABC[1].y-
ABC[2].y)**2)**0.5+((ABC[2].x-ABC[0].x)**2+(ABC[2].y-
ABC[0].y)**2)**0.5)
print(perimeter([point(0,0),point(0,4),point(3,0)]))
```

Результат:

```
12
```

Вычисление расстояния можно вынести в отдельную функцию:

```
from decpy import var

class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
```

```
def dist(A,B):
    return ((A.x-B.x)**2+(A.y-B.y)**2)**0.5

ABC = var()
perimeter = dist(ABC[0],ABC[1]) + dist(ABC[1],ABC[2]) +
dist(ABC[2],ABC[0])
print(perimeter([point(0,0),point(0,4),point(3,0)]))
```

Библиотека DecPy позволяет создавать комбинации ленивых индексов и атрибутов на неограниченную глубину, например:

```
A,B,C,D=var(3)
…
print(A[0].x[1])
print(B.x[1].y)
print(C[1][2].x.y)
print(D.x.y[1][2])
```

Выводы

Ленивые вычисления формируют особое мышление, «отделяющее алгебру от арифметики». Мы сначала объявляем (но не инициализируем) переменные, далее задаем формулы и только потом инициализируем переменные и «просим» Python вывести результат. Например, вернемся к первой задаче вычисления гипотенузы прямоугольника по двум катетам. Словесно алгоритм можно описать так:

1) Пусть `a` и `b` – катеты прямоугольного треугольника.
2) Гипотенуза c вычисляется по формуле …
3) Зададим `a` и `b` равными …
4) Вычислим `c`.

Программа именно в таком порядке отражает свойство человеческого мышления в начале приводить наиболее важную информацию (формулу вычисления), а потом – второстепенные детали (чему равны катеты).

Пока польза ленивых вычислений в рассмотренных примерах не видна. Их пока можно назвать программистской забавой. Но на основе ленивых вычислений будет построено всё декларативное программирование в следующих разделах книги. Именно там проявится вся мощь ленивых вычислений.

2. Фильтрация коллекций в стиле исчисления на кортежах

2.1. Фильтрация списков и множеств

Задача. Пусть дан список элементов. Сформировать новый список, элементы которого выбираются из исходного, если они удовлетворяют заданному условию.

В Python есть множество возможностей для решения этой задачи. Приведем наиболее распространенные в разных стилях программирования.

Структурный стиль программирования:

```
L=[10,20,30,40,50]
M=[]
for el in L:
    if el>15:
        M.append(el)
print(M)
```

Результат:

```
[20, 30, 40, 50]
```

Списочные вложения:

```
L=[10,20,30,40,50]
M=[el for el in L if el>15]
print(M)
```

Функциональный стиль:

```
L=[10,20,30,40,50]
M=list(filter(lambda el: el>15,L))
print(M)
```

Библиотека DecPy позволяет помещать условия в квадратные скобки:

```
from decpy import var

el=var()
L=[10,20,30,40,50]
M=var(L)[el>15]
print(M)
```

Эта программа, на мой взгляд, наиболее удачное решение. Она сочетает лаконичность функционального стиля с понятностью процедурного. В программе мы сделали три записи:

1) Объявили переменную `el`, смысл которой – элемент списка `L`:

```
el=var()
```

2) Упаковали список `L` в переменную `M` типа `var`:

```
M=var(L)
```

3) Написали условие отбора в квадратных скобках:

```
M=var(L)[el>15]
```

С этой программы начинается ***декларативное программирование*** – стиль (парадигма), в которой программист не пишет пошаговый алгоритм решения задачи, а описывает результат, который хочет получить. Интерпретатор (в нашем случае, библиотека DecPy) сама составит алгоритм.

Преимущество декларативного подхода перед функциональным в том, что в условии не нужно писать оператор `lambda`, смысл применения которого тяжело объяснить начинающим программистам и который засоряет код. Вместо:

```
lambda el: el>15,L
```

Мы написали:

```
el>15
```

Эта запись – не что иное, как ***ленивое выражение***. Если `el` не объявить как `var`, то тогда выражение `el` будет выполнено немедленно в квадратных скобках. Сейчас же выполнение условия будет отложено – вместо `el` будут в спрятанном библиотекой DecPy цикле подставляться элементы списка `L`.

Если в прошлом разделе ленивые формулы как стиль программирования – это просто остроумное экзотическое расширение языка Python, то здесь и далее ленивые вычисления – это основа и органическая часть декларативного программирования.

Заметим, что выборка `M` также объявлена как `var`, а значит, представляет собой ***ленивую коллекцию***. Убедимся в этом, добавив к `L` новый элемент и повторно выведя `M`:

```
from decpy import var

el=var()
L=[10,20,30,40,50]
M=var(L)[el>15]
print(M)
L.append(60)
print(M)
```

Результат:

```
[20, 30, 40, 50]
[20, 30, 40, 50, 60]
```

Получается, что `M` работает как понятие. Мы даем определение понятию `M`: список элементов `L`, бОльших 15. И это определение остается верным при

динамически меняющемся списке `L`. Понятие же представляет собой в терминах ленивых вычислений ***ленивый запрос***.

Приведем практический пример (заодно посмотрим, как работают ленивые атрибуты в запросах). Сделаем класс «Человек» с атрибутами «Имя» и «Год рождения». На экран будем выводить возраст человека. Для вычисления возраста еще введем глобальную переменную «Текущий год». Введем понятие «Молодежь» - люди, рожденные после 2000 года. Сделаем список людей, выведем его, а также список молодежи. Потом пополним список людей еще одним человеком… :

```
from decpy import var

curr_year=2025
class person:
    def __init__(self,name,year):
        self.name=name
        self.year=year
    def __repr__(self):
        return self.name+" "+str(curr_year-self.year)
p1=person("Василий",1990)
p2=person("Иван",2000)
p3=person("Мария",2001)
people=[p1,p2,p3]
el=var()
youth=var(people)[el.year>=2000]
print(people)
print(youth)
p4=person("Андрей",2024)
people.append(p4)
print(people)
print(youth)
```

Результат:

```
[Василий 35, Иван 25, Мария 24]
[Иван 25, Мария 24]
[Василий 35, Иван 25, Мария 24, Андрей 1]
[Иван 25, Мария 24, Андрей 1]
```

В нашем случае «молодежь» – это понятие. При пополнении списка людей список молодежи также пополняется. Нам нет необходимости писать новый запрос для молодежи, мы можем дать определение молодежи только один раз.

Получается, что **запросы в библиотеке DecPy работают как понятия**. К примеру с классом «Человек» мы еще неоднократно будем возвращаться. Пока же вернемся к примеру с простым списком. Можно отключить ленивость

выполнения запроса. Если нам нужна постоянная, а не динамически изменяемая при изменении исходных данных выборка, добавим к запросу пустые круглые скобки. Сравните:

Ленивая (динамическая) выборка:

```
from decpy import var

el=var()
L=[10,20,30,40,50]
M=var(L)[el>15]
print(M)
L.append(60)
print(M)
```

Результат:

```
[20, 30, 40, 50]
[20, 30, 40, 50, 60]
```

Постоянная выборка:

```
from decpy import var

el=var()
L=[10,20,30,40,50]
M=var(L)[el>15]()
print(M)
L.append(60)
print(M)
```

Результат:

```
[20, 30, 40, 50]
[20, 30, 40, 50]
```

Аналогично спискам можно осуществлять и фильтрацию множеств:

```
from decpy import var

el=var()
L={10,20,30,40,50}
M=var(L)[el>15]
print(M)
L.add(60)
print(M)
```

Результат:

```
{40, 50, 20, 30}

{40, 50, 20, 60, 30}
```

2.2. Запросы к запросам, множественные условия

Библиотека DecPy позволяет делать запросы к запросам:

```
from decpy import var

el=var()
L=[10,20,30,40,50]
M=var(L)[el>15]
P=M[el<45]
print(L)
print(M)
print(P)
```

Результат:

```
[10, 20, 30, 40, 50]
[20, 30, 40, 50]
[20, 30, 40]
```

Заметим, что `var` писать в определении `P` не обязательно.

Добавим элемент к исходному списку и убедимся, что свойство «ленивости» запросов сохраняется во всей цепочке запросов:

```
from decpy import var

el=var()
L=[10,20,30,40,50]
M=var(L)[el>15]
P=M[el<45]
print(L)
print(M)
print(P)
L.append(35)
print(L)
print(M)
print(P)
```

Результат:

```
[10, 20, 30, 40, 50]
[20, 30, 40, 50]
[20, 30, 40]

[10, 20, 30, 40, 50, 35]
[20, 30, 40, 50, 35]
[20, 30, 40, 35]
```

Условия отбора можно писать подряд в последовательности квадратных скобок. В определении `P` из предыдущей программы избавимся от промежуточного запроса `M`:

```
from decpy import var

el=var()
L=[10,20,30,40,50]
P=var(L)[el>15][el<45]
print(L)
print(P)
L.append(35)
print(L)
print(P)
```

Результат:

```
[10, 20, 30, 40, 50]
[20, 30, 40]

[10, 20, 30, 40, 50, 35]
[20, 30, 40, 35]
```

Чтобы не писать много квадратных скобок подряд, можно перечислить условия через запятую:

```
from decpy import var

el=var()
L=[10,20,30,40,50]
P=var(L)[el>15,el<45]
print(L)
print(P)
```

Перечислять условия через запятую нужно осторожно в случае, если элемент представляет собой составную коллекцию. Количество условий не должно равняться количеству составных частей элемента или превышать его. Иначе запросы будут работать по-другому (условия будут относиться не к элементу в целом, а к его составным частям, как в языке QBE). Пример мы рассмотрим в следующем разделе.

Рекомендую несколько условий соединять не запятыми, а операциями над множествами. Условие

```
el>15,el<45
```

можно понимать как пересечение (общую часть) списков с условием `el>15` и `el<45`:

```
from decpy import var

el=var()
L=[10,20,30,40,50]
P=var(L)[(el>15) & (el<45)]
print(L)
print(P)
```

Обратите внимание, что условия заключены в круглые скобки. Скобки обязательны, так как в Python операторы сравнения имеют меньший приоритет, чем операторы над множествами.

Оператору «`&`» соответствует «логическое и». «Логическому или» соответствует оператор «`|`»:

```
from decpy import var

el=var()
L=[10,20,30,40,50]
P=var(L)[(el<15) | (el>45)]
print(L)
print(P)
```

Результат:

```
[10, 20, 30, 40, 50]
[10, 50]
```

2.3. Групповые операции

Над коллекциями в Python есть групповые операции: поиск минимума (`min`), максимума (`max`), суммы (`sum`), количества элементов (`len`) и сортировка коллекции (`sorted`).

В библиотеке DecPy эти функции становятся ленивыми методами. К ним добавляются методы, взятые из языка запросов к базам данных SQL, - среднее количество (`avg`) и убирание дубликатов (`distinct`):

```
from decpy import var

L=var([30,20,50,10,40,20])
print(L)
print(L.min())
print(L.max())
print(L.avg())
print(L.sum())
print(L.len())
print(L.sorted())
print(L.distinct())
```

Результат:

```
[30, 20, 50, 10, 40, 20]
10
50
28.333333333333332
170
6
[10, 20, 20, 30, 40, 50]
[40, 10, 50, 20, 30]
```

Убедимся, что функции работают «лениво», создав соответствующие им переменные, а затем добавив к списку два элемента:

```
from decpy import var

L=var([30,20,50,10,40,20])
a=L.min()
b=L.max()
c=L.avg()
d=L.sum()
e=L.len()
f=L.sorted()
g=L.distinct()
print(L)
print(a)
print(b)
print(c)
print(d)
print(e)
print(f)
L.append(5)
L.append(60)
print(L)
print(a)
print(b)
print(c)
print(d)
print(e)
print(f)
print(g)
```

Результат:

```
[30, 20, 50, 10, 40, 20]
10
50
28.333333333333332
170
6
[10, 20, 20, 30, 40, 50]
[30, 20, 50, 10, 40, 20, 5, 60]
5
60
29.375
235
8
[5, 10, 20, 20, 30, 40, 50, 60]
[5, 40, 10, 50, 20, 60, 30]
```

Набор групповых операций можно расширить. Например, сумма элементов коллекции считается с помощью `sum`, но нет метода подсчета произведения. Посчитать произведение можно с помощью метода `reduce`, аргументом которого будет «стягивающая список» функция от двух аргументов:

```
from decpy import var

L=var([1,2,3,4,5])
print(L.sum())
x,y=var(2)
print(L.reduce(x*y))
```

Результат:

```
15

120
```

Убедимся в «ленивости» вычислений, введя соответствующие переменные и добавив элемент к исходному списку:

```
from decpy import var

L=var([1,2,3,4,5])
a=L.sum()
x,y=var(2)
b=L.reduce(x*y)
print(L)
print(a)
print(b)
L.append(6)
print(L)
print(a)
print(b)
```

Результат:

```
[1, 2, 3, 4, 5]
15
120

[1, 2, 3, 4, 5, 6]
21
720
```

Название метода `reduce` взято от одноименной функции из библиотеки функционального программирования.

2.4. Новый список на основе вычислений над элементами исходного

Задача. На основе исходного списка сформировать список квадратов его элементов.

Как и в случае с фильтрацией, приведем три решения на основе языковых конструкций Python в разных стилях программирования.

Структурный стиль:

```
L=[1,2,3,4,5]
M=[]
for el in L:
    M.append(el**2)
print(L)
print(M)
```

Результат:

```
[1, 2, 3, 4, 5]
[1, 4, 9, 16, 25]
```

Списочные выражения:

```
L=[1,2,3,4,5]
M=[el**2 for el in L]
print(L)
print(M)
```

Функциональный стиль:

```
L=[1,2,3,4,5]
M=list(map(lambda el: el**2,L))
print(L)
print(M)
```

Библиотека DecPy позволяет написать программу в декларативном стиле:

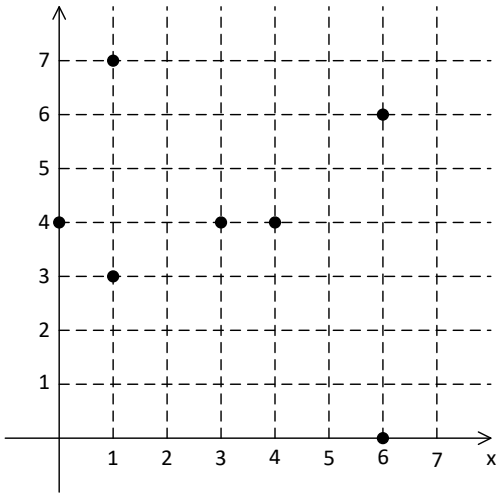
```
from decpy import var

L=[1,2,3,4,5]
el=var()
M=var(L)[el**2]
print(L)
print(M)
```

Как и в случае с фильтрацией, декларативный стиль выглядит привлекательнее других.

2.5. Выборки из составной коллекции

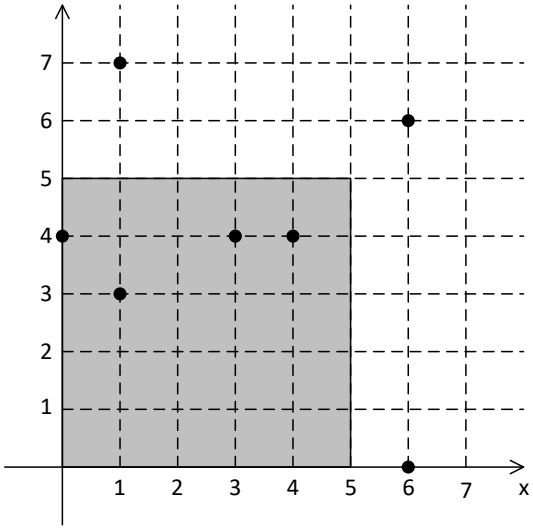
Рассмотрим теперь коллекции, элементы которых сами имеют внутреннюю структуру, например, зададим список точек на плоскости следующим образом:

```
L=var([(6,0),(1,7),(0,4),(4,4),(1,3),(6,6),(3,4)])
```

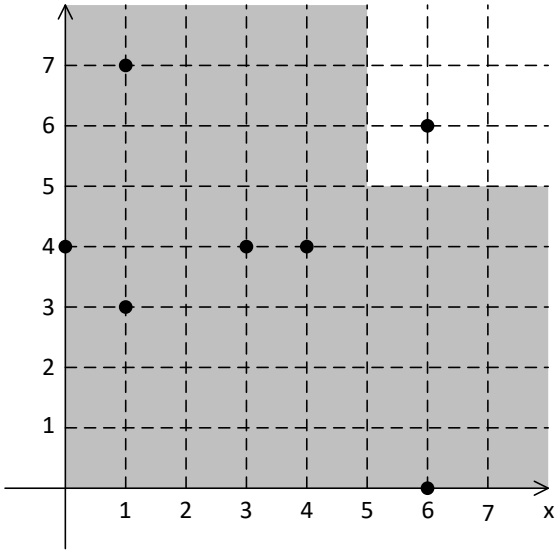


Здесь точке соответствует пара координат. Сделаем три выборки точек:

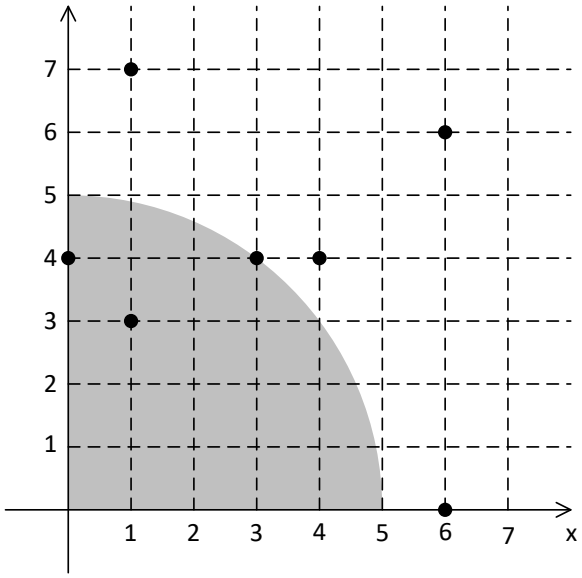
1) точки, попадающие в квадрат размером 5 на 5;



2) точки, хотя бы одна координата которых меньше или равна 5;



3) Точки, расстояние которых до центра координат меньше или равно 5.



Напишем программу для трех выборок:

```
from decpy import var

L=var([(6,0),(1,7),(0,4),(4,4),(1,3),(6,6),(3,4)])
print(L)
el=var()
M=L[(el[0]<=5) & (el[1]<=5)]
print(M)
P=L[(el[0]<=5) | (el[1]<=5)]
print(P)
R=L[el[0]**2+el[1]**2<=25]
print(R)
```

Результат:

```
[(6,0), (1,7), (0,4), (4,4), (1,3), (6,6), (3,4)]
[(0,4), (4,4), (1,3), (3,4)]
[(6,0), (1,7), (0,4), (4,4), (1,3), (3,4)]
[(0,4), (1,3), (3,4)]
```

Заметим, что в этом примере используются ленивые индексы.

Можно создать отдельный класс «Точка» с координатами `x` и `y`. В этом случае будем использовать ленивые атрибуты:

```
from decpy import var

class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
    def __repr__(self):
        return "("+str(self.x)+","+str(self.y)+")"

L=var([point(6,0),point(1,7),point(0,4),point(4,4),point(1,
3),point(6,6),point(3,4)])
print(L)
el=var()
M=L[(el.x<=5) & (el.y<=5)]
print(M)
P=L[(el.x<=5) | (el.y<=5)]
print(P)
R=L[el.x**2+el.y**2<=25]
print(R)
```

Вычисление расстояния от начала координат можно поместить в отдельный метод `abs` (точку с математической точки зрения можно рассматривать как вектор, чью длину мы вычисляем). Чтобы сделать метод `abs` ленивым, используем `lazyfun`:

```
from decpy import var,lazyfun

class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
    def __repr__(self):
        return "("+str(self.x)+","+str(self.y)+")"
    def abs(self):
        return (self.x**2+self.y**2)**0.5
```

```
L=var([point(6,0),point(1,7),point(0,4),point(4,4),point(1,
3),point(6,6),point(3,4)])
print(L)
el=var()
R=L[lazyfun(point.abs)(el)<=5]
print(R)
```

Точки могут иметь самостоятельное значение, помимо включения их в список. Создадим точки как переменные, а потом добавим их в список:

```
from decpy import var,lazyfun

class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
    def __repr__(self):
        return "("+str(self.x)+","+str(self.y)+")"
    def abs(self):
        return (self.x**2+self.y**2)**0.5

A=point(6,0)
B=point(1,7)
C=point(0,4)
D=point(4,4)
E=point(1,3)
F=point(6,6)
G=point(3,4)
L=var([A,B,C,D,E,F,G])
print(L)
el=var()
M=L[(el.x<=5) & (el.y<=5)]
print(M)
P=L[(el.x<=5) | (el.y<=5)]
print(P)
R=L[lazyfun(point.abs)(el)<=5]
print(R)
```

2.6. Запросы к классам

Продолжим предыдущий пример – выборки из коллекции точек.

Если запрос производится ко всем без исключения точкам, то создание новых точек потребует двух строк кода:

```
...
H=point(8,9)
L.append(H)
```

В объектно-ориентированных базах данных есть три понятия: ***класс***, ***экземпляры*** класса и ***экстент*** – коллекция, в которой сохраняются экземпляры. В нашем случае `point` – это класс, точка `H` – экземпляр класса, а список `L` – экстент. В большинстве случаев разделение на класс и экстент излишне и требует дополнительных строк кода.

Библиотека DecPy предоставляет декоратор `queryclass`, который автоматически создает экстент для хранения экземпляров класса. Причем экстент имеет то же самое имя, что и класс. При его использовании нет необходимости создавать список, а запросы можно писать к самому классу:

```
from decpy import var,queryclass,lazyfun

@queryclass
class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
    def __repr__(self):
        return "("+str(self.x)+","+str(self.y)+")"
    def abs(self):
        return (self.x**2+self.y**2)**0.5

A=point(6,0)
B=point(1,7)
C=point(0,4)
D=point(4,4)
E=point(1,3)
F=point(6,6)
G=point(3,4)
print(point)
el=var()
M=point[(el.x<=5) & (el.y<=5)]
print(M)
P=point[(el.x<=5) | (el.y<=5)]
print(P)
R=point[lazyfun(point.abs)(el)<=5]
print(R)
```

Если точки по отдельности не имеют значения, то им можно не давать имена:

```
from decpy import var,queryclass,lazyfun

@queryclass
class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
    def __repr__(self):
        return "("+str(self.x)+","+str(self.y)+")"
    def abs(self):
        return (self.x**2+self.y**2)**0.5

point(6,0)
point(1,7)
point(0,4)
point(4,4)
point(1,3)
point(6,6)
point(3,4)
print(point)
el=var()
M=point[(el.x<=5) & (el.y<=5)]
print(M)
P=point[(el.x<=5) | (el.y<=5)]
print(P)
R=point[lazyfun(point.abs)(el)<=5]
print(R)
```

Преобразуем код для предметной области «Люди», добавив декоратор `queryclass`. Посмотрим, как работает выборка молодежи:

```
from decpy import var,queryclass

curr_year=2025
@queryclass
class person:
    def __init__(self,name,year):
        self.name=name
        self.year=year
    def __repr__(self):
        return self.name+" "+str(curr_year-self.year)

person("Василий",1990)
person("Иван",2000)
person("Мария",2001)
el=var()
youth= person[el.year>=2000]
```

```
print(person)
print(youth)
p4=person("Андрей",2024)
print(person)
print(youth)
```

Результат:

```
[Василий 35, Иван 25, Мария 24]
[Иван 25, Мария 24]
[Василий 35, Иван 25, Мария 24, Андрей 1]
[Иван 25, Мария 24, Андрей 1]
```

Если класс не предусматривает сложной внутренней структуры (например, является просто набором данных разных типов) и методов обработки данных, то писать класс может быть нецелесообразно. Программа с точкой как парой чисел явно по лаконичности выигрывает у программы, где точка – это экземпляр класса. Единственное преимущество у программы с классом – это обращение к координатам по их имени – `x` и `y` (ленивые атрибуты), в отличие от программы с парой чисел, где к координатам приходится обращаться по их номеру (ленивые индексы).

Чтобы не описывать точки как класс, но в то же время пользоваться ленивыми атрибутами, в библиотеке DecPy предусмотрена функция `table`, фактически создающая новый класс пересчислением в её аргументе названий атрибутов:

```
from decpy import var, table

point=table("x","y")
point(6,0)
point(1,7)
point(0,4)
point(4,4)
point(1,3)
point(6,6)
point(3,4)
print(point)
el=var()
M=point[(el.x<=5) & (el.y<=5)]
print(M)
P=point[(el.x<=5) | (el.y<=5)]
print(P)
R=point[el.x**2+el.y**2<=25]
print(R)
```

Заметим, что при использовании `table` нам пришлось убрать метод вычисления расстояния от начала координат `abs` и прописать формулу вычисления расстояния в запросе.

В разных информационных технологиях `table` называется по-разному. В реляционных базах данных элементы `table` – кортежи, в языке искусственного интеллекта Prolog – предикаты. В библиотеке DecPy используется нейтральное название - таблица (`table`).

2.7. Проекции и вычисления

С помощью квадратных скобок можно выбирать отдельные атрибуты или делать вычисления. Например, на основе множества точек сформируем по отдельности множество координат `x`, множество координат `y` и множество расстояний до центра координат:

```
from decpy import var,table

point=table("x","y")
point(6,0)
point(1,7)
point(0,4)
point(4,4)
point(1,3)
point(6,6)
point(3,4)
print(point)
el=var()
M=point[el.x]
print(M)
P=point[el.y]
print(P)
R=point[el.x**2+el.y**2]
print(R)
```

Результат:

```
{(6, 6), (0, 4), (6, 0), (1, 3), (1, 7), (3, 4), (4, 4)}
{0, 1, 3, 4, 6}
{0, 3, 4, 6, 7}
{3.1622776601683795, 4.0, 5.656854249492381, 5.0,
7.0710678118654755, 8.48528137423857, 6.0}
```

2.8. Групповые операции над составной коллекцией

Соединим идеи предыдущих глав в одной задаче.

Отсортируем точки по удаленности от начала координат и найдем самую удаленную точку с помощью групповых операций:

```
from decpy import var,table

point=table("x","y")
point(6,0)
point(1,7)
point(0,4)
point(4,4)
point(1,3)
point(6,6)
point(3,4)
print(point)
el=var()
Q=point[el].max((el.x**2+el.y**2)**0.5)
print(Q)
S=point[el].sorted((el.x**2+el.y**2)**0.5)
print(S)
```

Результат:

```
(6, 6)
[(1, 3), (0, 4), (3, 4), (4, 4), (6, 0), (1, 7), (6, 6)]
```

Выводы

Заметим, что во всех запросах этого раздела фигурировала переменная, которую я назвал `el` (сокращение от слова «element»). Назвать её можно было бы и по-другому, но смысл её остался бы прежним: переменная - элемент исходной коллекции. В терминах реляционных баз данных переменная – кортеж. Есть раздел математики, изучающий операции над переменными-кортежами. Он назвается ***исчисление на кортежах***. На этом исчислении основан наиболее распространенный язык работы с базами данных - SQL.

В этом разделе мы рассмотрели не все возможности, аналогичные возможностям SQL. Например, мы не привели запросы к нескольким коллекциям. Их мы изучим в следующем разделе.

3. Запросы в стиле исчисления на доменах

Уже упоминалось, что во всех запросах из предыдущей главы использовалась переменная `el`, смысл которой – переменная-элемент перебираемой коллекции. И постоянное её применение начинает раздражать так же, как и использование `lambda` в функциональном программировании. Можно ли обойтись без неё? Можно, если в запросе использовать несколько переменных под каждый атрибут. В основе таких запросов лежит раздел математики ***исчисление на доменах***. Оно реализовано в библиотеке DecPy, и библиотека сама определяет тип исчисления по виду запроса.

3.1. Альтернатива исчислению на кортежах

Для сравнения приведем несколько запросов к списку точек на плоскости в исчислении на доменах и исчислении на кортежах:

Исчисление на доменах:

```
from decpy import var

L=var([(6,0),(1,7),(0,4),(4,4),(1,3),(6,6),(3,4)])
x,y,d=var(3)
print(L)
print(L[x, None])
print(L[None, y])
print(L[x<=5, y<=5])
print(L[x, y, (x<=5) | (y<=5)])
print(L[x, y, x**2+y**2<=25])
```

Исчисление на кортежах:

```
from decpy import var,table

point=table("x","y")
point(6,0)
point(1,7)
point(0,4)
point(4,4)
point(1,3)
point(6,6)
point(3,4)
print(point)
el=var()
print(point[el.x])
print(point[el.y])
print(point[(el.x<=5) & (el.y<=5)])
print(point[(el.x<=5) | (el.y<=5)])
print(point[el.x**2+el.y**2<=25])
```

Результаты обеих программ одинаковы:

```
[(6, 0), (1, 7), (0, 4), (4, 4), (1, 3), (6, 6), (3, 4)]
{0, 1, 3, 4, 6}
{0, 3, 4, 6, 7}
{(4, 4), (1, 3), (3, 4), (0, 4)}
{(4, 4), (0, 4), (3, 4), (1, 7), (6, 0), (1, 3)}
{(1, 3), (3, 4), (0, 4)}
```

Заметим, что в исчислении на доменах перечисляются переменные, соответствующие всем атрибутам исходной таблицы, а если какой-либо атрибут не нужен, на его месте пишется `None`. После перечисления всех атрибутов, возможно, идут дополнительные вычисления и условия.

На основе исчисления на кортежах работает язык запросов к базам данных SQL, на основе исчисления на доменах – язык QBE. В чистом виде QBE не получил распространения, но он лежит в основе графических средств (мастеров) создания запросов. Так, в настольной базе данных Access, входящей в Microsoft Office, в режиме мастера запрос конструируется как QBE, а в текстовом виде – как SQL.

3.2. Запросы с группировками

В следующих главах этого раздела покажем возможности библиотеки DecPy для выборок из таблиц по типу запросов к базам данных. В основном будем использовать исчисление на доменах. Лучше всего продемонстрировать возможности на торговой базе данных. Создадим таблицу «Сделки», содержащую следующие колонки: продавец, покупатель, товар, цена, количество:

```
from decpy import var,table
sales=table("salesman","buyer","good","price","amount")
sales("Тысяча мелочей","Петров","Шило",10,5)
sales("Тысяча мелочей","Петров","Мыло",15,10)
sales("Тысяча мелочей","Сидоров","Веревка",50,1)
sales("Хозтовары","Сидоров","Мыло",20,30)
sales("Хозтовары","Петров","Мыло",12,10)
for s in sales:
    print(s)
```

Результат:

```
('Тысяча мелочей', 'Сидоров', 'Веревка', 50, 1)
('Хозтовары', 'Петров', 'Мыло', 12, 10)
('Тысяча мелочей', 'Петров', 'Мыло', 15, 10)
('Хозтовары', 'Сидоров', 'Мыло', 20, 30)
('Тысяча мелочей', 'Петров', 'Шило', 10, 5)
```

Запрос №1. Вывести покупателей и покупаемые ими товары.

Напишем запрос в стиле исчисления на кортежах:

```
from decpy import var,table
sales=table("salesman","buyer","good","price","amount")
sales("Тысяча мелочей","Петров","Шило",10,5)
sales("Тысяча мелочей","Петров","Мыло",15,10)
sales("Тысяча мелочей","Сидоров","Веревка",50,1)
sales("Хозтовары","Сидоров","Мыло",20,30)
sales("Хозтовары","Петров","Мыло",12,10)
el=var()
S=sales[el.buyer,el.good]
for s in S:
    print(s)
```

Результат:

```
('Петров', 'Шило')
('Петров', 'Мыло')
('Сидоров', 'Веревка')
('Сидоров', 'Мыло')
```

Аналогичный запрос можно написать в стиле исчисления на доменах:

```
from decpy import var,table
sales=table("salesman","buyer","good","price","amount")
sales("Тысяча мелочей","Петров","Шило",10,5)
sales("Тысяча мелочей","Петров","Мыло",15,10)
sales("Тысяча мелочей","Сидоров","Веревка",50,1)
sales("Хозтовары","Сидоров","Мыло",20,30)
sales("Хозтовары","Петров","Мыло",12,10)
buyer,good=var(2)
S=sales[None,buyer,good,None,None]
for s in S:
    print(s)
```

Запрос №2. Для каждой пары «покупатель / товар» составить списки магазинов, в которых были совершены эти покупки.

Воспользуемся новым для нас методом `group` из библиотеки DecPy, чтобы сгруппировать записи по покупателям и товарам:

```
from decpy import var,table
...
salesman,buyer,good=var(3)
S=sales[salesman,buyer.group(),good.group(),None,None]
for s in S:
    print(s)
```

Результат:

```
(('Тысяча мелочей',), 'Петров', 'Шило')
(('Хозтовары',), 'Сидоров', 'Мыло')
(('Тысяча мелочей',), 'Сидоров', 'Веревка')
(('Хозтовары', 'Тысяча мелочей'), 'Петров', 'Мыло')
```

Видно, что магазины теперь помещены в списки и Петров купил мыло сразу в двух магазинах. Эти вложенные списки можно обрабатывать с помощью групповых операций.

Запрос демонстрирует группировку сразу по двум атрибутам. Для дальнейших рассуждений упростим задачу и сделаем группировку по одному атрибуту.

Запрос №3. В каких магазинах покупатели делали свои покупки?

```
from decpy import var,table
...
salesman,buyer=var(2)
S=sales[salesman,buyer.group(),None,None,None]
for s in S:
    print(s)
```

Результат:

```
(('Тысяча мелочей', 'Хозтовары', 'Тысяча мелочей'),
'Петров')
(('Хозтовары', 'Тысяча мелочей'), 'Сидоров')
```

Заметим, что сформированные кортежи магазинов содержат дубликаты. Так, Петров заходил в «Тысячу мелочей» два раза. На основе этого запроса можно сделать два следующих.

Запрос №4. Убрать дубликаты из предыдущего запроса.

Воспользуемся методом `distinct`:

```
from decpy import var,table
...
salesman,buyer=var(2)
S=sales[salesman.distinct(),buyer.group(),None,None,None]
for s in S:
    print(s)
```

Результат:

```
(('Хозтовары', 'Тысяча мелочей'), 'Петров')
(('Хозтовары', 'Тысяча мелочей'), 'Сидоров')
```

Запрос №5. Посчитать количество покупок, сделанных каждым покупателем.

Добавим к запросу №3 метод `len`:

```
from decpy import var,table
...
salesman,buyer=var(2)
S=sales[salesman.len(),buyer.group(),None,None,None]
for s in S:
    print(s)
```

Результат:

```
(3, 'Петров')
(2, 'Сидоров')
```

Такой порядок колонок может быть неудобным. Переставим их местами с помощью кода, выделенного жирным:

```
from decpy import var,table
...
salesman,buyer=var(2)
S=sales[salesman.len(),buyer.group(),None,None,None]
T,X,Y=var(3)
T[X,Y]|=S[Y,X]
for t in T:
    print(t)
```

Результат:

```
('Сидоров', 2)
('Петров', 3)
```

Работа перестановки интуитивно понятна. Причина, по которой мы здесь использовали оператор «|=», а не «=», будет изложена в следующем разделе.

Решим еще одну задачу с группировками.

Задача. Для каждого покупателя посчитать общую сумму потраченных на покупки денег.

Решим задачу пошагово.

Запрос №6. Добавить колонку «Стоимость», которая расчитывается как произведение цены на количество:

```
from decpy import var,table
...
salesman,buyer,good,price,amount=var(5)
S=sales[salesman,buyer,good,price,amount,price*amount]
for s in S:
    print(s)
```

Результат:

```
('Тысяча мелочей', 'Петров', 'Шило', 10, 5, 50)
('Тысяча мелочей', 'Петров', 'Мыло', 15, 10, 150)
('Хозтовары', 'Петров', 'Мыло', 12, 10, 120)
('Хозтовары', 'Сидоров', 'Мыло', 20, 30, 600)
('Тысяча мелочей', 'Сидоров', 'Веревка', 50, 1, 50)
```

Запрос №7. Для каждого продавца посчитать общую стоимость сделок.

К предыдущему запросу применим группировку, уберем лишние колонки и применим метод `sum`:

```
from decpy import var,table
sales=table("salesman","buyer","good","price","amount")
sales("Тысяча мелочей","Петров","Шило",10,5)
sales("Тысяча мелочей","Петров","Мыло",15,10)
sales("Тысяча мелочей","Сидоров","Веревка",50,1)
sales("Хозтовары","Сидоров","Мыло",20,30)
sales("Хозтовары","Петров","Мыло",12,10)
salesman,buyer,good,price,amount,cost=var(6)
S=sales[salesman,buyer,good,price,amount,price*amount]
       [None,buyer.group(),None,None,None,cost.sum()]
for s in S:
    print(s)
```

Результат:

```
('Петров', 320)
('Сидоров', 650)
```

3.3. Запросы с кванторами

В Python для обработки коллекций используются кванторы `all` и `any`. С помощью квантора `all` можно определить, все ли элементы коллекции удовлетворяют заданному условию. С помощью квантора `any` можно определить, есть ли хотя бы один элемент, удовлетворяющий заданному условию.

В библиотеке DecPy есть аналогичные кванторы `All` и `Any`, являющиеся ленивыми методами. Рассмотрим применение этих кванторов в запросах. Для этого сделаем таблицу, аналогичную таблице из предыдущей главы:

```
from decpy import var,table
sales=table("salesman","buyer","good","price","amount")
sales("Рога и копыта","Иванов","Рога",10,1)
sales("Рога и копыта","Иванов","Рога",15,2)
sales("Рога и копыта","Петров","Рога",10,1)
sales("Рога и копыта","Петров","Копыта",5,4)
sales("Охотник","Сидоров","Патроны",10,20)
```

```
for s in sales:
    print(s)
```

Результат:

```
('Рога и копыта', 'Петров', 'Копыта', 5, 4)
('Рога и копыта', 'Иванов', 'Рога', 10, 1)
('Рога и копыта', 'Петров', 'Рога', 10, 1)
('Охотник', 'Сидоров', 'Патроны', 10, 20)
('Рога и копыта', 'Иванов', 'Рога', 15, 2)
```

Запрос №1. Для каждого покупателя составить список покупок.

Применим группировку `group`:

```
from decpy import var,table
...
buyer,good=var(2)
S=sales[None,buyer.group(),good,None,None]
for s in S:
    print(s)
```

Результат:

```
('Иванов', ('Рога', 'Рога'))
('Петров', ('Копыта', 'Рога'))
('Сидоров', ('Патроны',))
```

Запрос №2. Вывести покупателей, которые покупали только рога:

Применим квантор `All`:

```
from decpy import var,table
...
buyer,good,el=var(3)
S=sales[None,buyer.group(),good.All(el=="Рога"),None,None]
for s in S:
    print(s)
```

Результат:

```
('Иванов', ('Рога', 'Рога'))
```

Для простых условий, в которых атрибут сопоставляется с образцом, можно писать только образец. Вместо

```
S=sales[None,buyer.group(),good.All(el=="Рога"),None,None]
```

можно написать кратко:

```
S=sales[None,buyer.group(),good.All("Рога"),None,None]
```

Уберем списки товаров из результата запроса, оставив только покупателей:

```
from decpy import var,table
...
buyer,good=var(2)
S=sales[None,buyer.group(),good.All("Рога"),None,None][buye
r,None]
for s in S:
    print(s)
```

Результат:

```
Иванов
```

Запрос №2. Вывести покупателей, которые хотя бы раз купили рога.

Заменим квантор `All` на `Any`:

```
from decpy import var,table
sales=table("salesman","buyer","good","price","amount")
sales("Рога и копыта","Иванов","Рога",10,1)
sales("Рога и копыта","Иванов","Рога",15,2)
sales("Рога и копыта","Петров","Рога",10,1)
sales("Рога и копыта","Петров","Копыта",5,4)
sales("Охотник","Сидоров","Патроны",10,20)
buyer,good=var(2)
S=sales[None,buyer.group(),good.Any("Рога"),None,None][buye
r,None]
for s in S:
    print(s)
```

Результат:

```
Петров
Иванов
```

В дополнение к Иванову мы получили еще и Петрова, который покупал не только рога, но и копыта.

Запрос с квантором `Any` можно было бы написать и без использования кванторов, путем сопоставления с образцом.

```
from decpy import var,table
...
buyer,good=var(2)
S=sales[None,buyer,good,None,None,good=="Рога"]
for s in S:
    print(s)
```

Результат:

```
('Иванов', 'Рога')
('Петров', 'Рога')
```

Условие можно написать не отдельно, а прямо у нужного атрибута:

```
S=sales[None,buyer,good=="Рога",None,None]
```

Или еще проще, если сравнение – равенство:

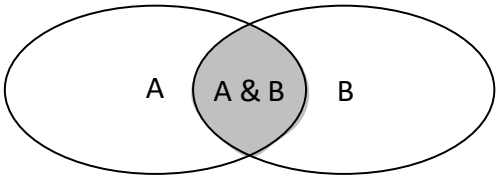
```
S=sales[None,buyer,"Рога",None,None]
```

3.4. Объединение однотипных запросов

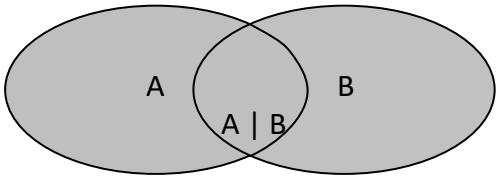
Однотипные запросы можно объединять. Таблицы из двух предыдущих глав имеют одинаковую структуру. В них есть общие покупатели. Соответственно, можно написать следующие запросы:

1) Вывести единый список покупателей из двух таблиц.
2) Вывести общих покупаталей.
3) Вывести покупателей из первой таблицы, которых нет во второй.
4) Вывести покупателей из второй таблицы, которых нет в первой.
5) Вывести покупателей, которые есть только в одной таблице из двух.

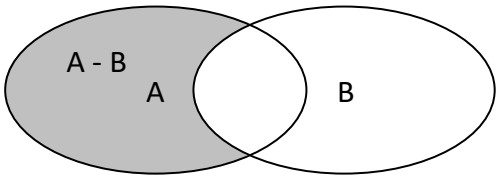
Эти задачи можно изобразить в виде ***кругов Эйлера***. Им соответствуют операторы Python для работы с множествами:



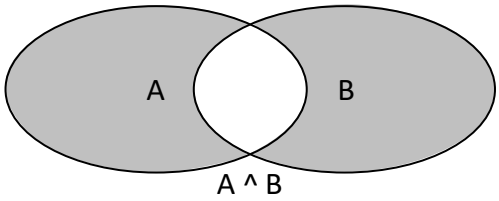
Пересечение



Объединение



Разность



Симметрическая разность

Эти операторы можно применять к таблицам `table` из библиотеки DecPy:

```
from decpy import var,table

salesA=table("salesman","buyer","good","price","amount")
salesA("Тысяча мелочей","Петров","Шило",10,5)
salesA("Тысяча мелочей","Петров","Мыло",15,10)
salesA("Тысяча мелочей","Сидоров","Веревка",50,1)
salesA("Хозтовары","Сидоров","Мыло",20,30)
salesA("Хозтовары","Петров","Мыло",12,10)
salesA("Охотник","Смирнов","Рогатка",10,20)

salesB=table("salesman","buyer","good","price","amount")
salesB("Рога и копыта","Иванов","Рога",10,1)
salesB("Рога и копыта","Иванов","Рога",15,2)
salesB("Рога и копыта","Петров","Рога",10,1)
salesB("Рога и копыта","Петров","Копыта",5,4)
salesB("Охотник","Сидоров","Патроны",10,20)

buyer=var()
A=salesA[None,buyer,None,None,None]
print(A)
B=salesB[None,buyer,None,None,None]
print(B)
print(A | B)
print(A & B)
print(A - B)
print(B - A)
print(B ^ A)
```

Результат:

```
{'Смирнов', 'Сидоров', 'Петров'}
{'Сидоров', 'Петров', 'Иванов'}
{'Иванов', 'Смирнов', 'Сидоров', 'Петров'}
{'Сидоров', 'Петров'}
{'Смирнов'}
{'Иванов'}
{'Смирнов', 'Иванов'}
```

Все запросы в этой главе мы делали к `table`, но заметим, что все они применимы и в других случаях, когда, например, коллекция задана классом с декоратором `@queryclass` или множеством, помещенным в ленивую переменную:

```
from decpy import var

salesA=var({("salesman","buyer","good","price","amount"),
    ("Тысяча мелочей","Петров","Шило",10,5),
    ("Тысяча мелочей","Петров","Мыло",15,10),
    ("Тысяча мелочей","Сидоров","Веревка",50,1),
    ("Хозтовары","Сидоров","Мыло",20,30),
    ("Хозтовары","Петров","Мыло",12,10),
    ("Охотник","Смирнов","Рогатка",10,20)})

salesB=var({("salesman","buyer","good","price","amount"),
    ("Рога и копыта","Иванов","Рога",10,1),
    ("Рога и копыта","Иванов","Рога",15,2),
    ("Рога и копыта","Петров","Рога",10,1),
    ("Рога и копыта","Петров","Копыта",5,4),
    ("Охотник","Сидоров","Патроны",10,20)})

buyer=var()
A=salesA[None,buyer,None,None,None]
print(A)
B=salesB[None,buyer,None,None,None]
print(B)
print(A | B)
```

3.5. Запросы к нескольким таблицам

Очень часто в базах данных делаются выборки из нескольких таблиц. Научимся писать такие запросы с помощью библиотеки DecPy.

Рассмотрим две таблицы:

1) Таблицу «Человек», хранящую имя и год рождения.
2) Таблицу «Собака», хранящую имя, год рождения и имя владельца.

```
from decpy import var, table

person=table("name","year")
person("Иванов",2000)
person("Петров",2005)
person("Сидоров",2010)
print(person)

dog=table("name","year","owner")
dog("Шарик",2010,"Иванов")
dog("Жучка",2015,"Петров")
dog("Бобик",2020,"Иванов")
print(dog)
```

Результат:

```
{('Иванов', 2000), ('Сидоров', 2010), ('Петров', 2005)}
{('Шарик', 2010, 'Иванов'), ('Бобик', 2020, 'Иванов'),
('Жучка', 2015, 'Петров')}
```

Соединим эти таблицы вместе. В реляционных базах данных есть операция ***декартово произведение***, при которой каждый элемент одной коллекции соединяется с каждым элементом второй коллекции. В библиотеке DecPy этой операции соответствует обычное умножение «*»:

```
from decpy import var, table

person=table("name","year")
person("Иванов",2000)
person("Петров",2005)
person("Сидоров",2010)

dog=table("name","year","owner")
dog("Шарик",2010,"Иванов")
dog("Жучка",2015,"Петров")
dog("Бобик",2020,"Иванов")

L=person*dog
for el in L:
    print(el)
```

Результат:

```
(('Петров', 2005), ('Жучка', 2015, 'Петров'))
(('Иванов', 2000), ('Жучка', 2015, 'Петров'))
(('Сидоров', 2010), ('Бобик', 2020, 'Иванов'))
(('Сидоров', 2010), ('Шарик', 2010, 'Иванов'))
(('Петров', 2005), ('Шарик', 2010, 'Иванов'))
(('Иванов', 2000), ('Шарик', 2010, 'Иванов'))
(('Петров', 2005), ('Бобик', 2020, 'Иванов'))
(('Иванов', 2000), ('Бобик', 2020, 'Иванов'))
(('Сидоров', 2010), ('Жучка', 2015, 'Петров'))
```

В исходных таблицах по три строчки. Поскольку декартово произведение соединяет все строки одной таблицы со всеми строками другой, мы получаем 3*3=9 строк в ответе.

Декартово произведение само по себе редко бывает нужным. Нас интересуют пары «собаковод - собака». Поэтому применим фильтрацию, чтобы собаки выводились на экран именно со своими владельцами:

```
from decpy import var, table
...
el=var()
L=(person*dog)[el[0].name==el[1].owner]
for el in L:
    print(el)
```

Результат:

```
(('Иванов', 2000), ('Шарик', 2010, 'Иванов'))
(('Иванов', 2000), ('Бобик', 2020, 'Иванов'))
(('Петров', 2005), ('Жучка', 2015, 'Петров'))
```

Запрос в стиле исчисления на кортежах неизящен. Поскольку результат образует пару, человеку соответствует `el[0]`, а собаке – `el[1]`. Перепишем запрос в стиле исчисления на доменах:

```
from decpy import var, table
...
p,d=var(2)
L=(person*dog)[p,d,p.name==d.owner]
for el in L:
    print(el)
```

Здесь в квадратных скобках сперва перечисляются псевдонимы таблиц, участвующих в произведении, то есть `person` соответствует `p`, а `dog` – `d`, а потом пишется условие связи таблиц.

Оператор декартова произведения «*» сохраняет внутреннюю структуру исходных таблиц – результат является кортежем кортежей. Посмотрите на одну из строк результата:

```
(('Иванов', 2000), ('Шарик', 2010, 'Иванов'))
```

Здесь явно сгруппированы атрибуты человека и атрибуты собаки. Сохранение такой группировки может быть полезно, но оно избыточно – имя владельца «Иванов» повторяется у собаки. Группировку можно убрать, используя второй оператор декартова произведения из библиотеки DecPy – «**»:

```
from decpy import var, table
...
L=person**dog
for el in L:
    print(el)
```

Результат:

```
('Сидоров', 2010, 'Шарик', 2010, 'Иванов')
('Петров', 2005, 'Жучка', 2015, 'Петров')
('Сидоров', 2010, 'Бобик', 2020, 'Иванов')
('Иванов', 2000, 'Бобик', 2020, 'Иванов')
('Иванов', 2000, 'Шарик', 2010, 'Иванов')
('Иванов', 2000, 'Жучка', 2015, 'Петров')
('Петров', 2005, 'Бобик', 2020, 'Иванов')
('Сидоров', 2010, 'Жучка', 2015, 'Петров')
('Петров', 2005, 'Шарик', 2010, 'Иванов')
```

Обратите внимание, что при использовании оператора «**» отсутствует внутреннее разделение атрибутов человека и собаки. Теперь нужно связать собак с их владельцами. Для этого в режиме исчисления на доменах потребуется больше переменных:

```
from decpy import var, table
...
name1,name2,year1,year2,owner=var(5)
L=(person**dog)[name1,year1,name2,year2,owner,name1==owner]
for el in L:
    print(el)
```

Результат:

```
('Петров', 2005, 'Жучка', 2015, 'Петров')
('Иванов', 2000, 'Бобик', 2020, 'Иванов')
('Иванов', 2000, 'Шарик', 2010, 'Иванов')
```

Остается убрать из результата одинаковые колонки. Можно добавить к запросу еще одни квадратные скобки и, чтобы не выводить последний атрибут, поставить на его месте None. Но есть средство изящнее. Мы обойдемся одними квадратными скобками, просто в совпадающих колонках напишем одинаковые переменные. Вместо:

```
L=(person**dog)[name1,year1,name2,year2,owner,name1==owner]
```

Напишем:

```
L=(person**dog)[name1,year1,name2,year2,name1]
```

Получим готовую программу:

```
from decpy import var, table

person=table("name","year")
person("Иванов",2000)
person("Петров",2005)
person("Сидоров",2010)
```

```
dog=table("name","year","owner")
dog("Шарик",2010,"Иванов")
dog("Жучка",2015,"Петров")
dog("Бобик",2020,"Иванов")

name1,name2,year1,year2,owner=var(5)
L=(person**dog)[name1,year1,name2,year2,name1]
for el in L:
    print(el)
```

Результат:

```
('Петров', 2005, 'Жучка', 2015)
('Иванов', 2000, 'Бобик', 2020)
('Иванов', 2000, 'Шарик', 2010)
```

Обратите внимание, что библиотека DecPy из повторяющихся колонок оставила одну, убрав дубликат.

3.6. Гибридный вариант запроса исчислений на кортежах и доменах

Как вы, наверное, заметили, программа в декларативном стиле с помощью библиотеки DecPy состоит из последовательности выборок с помощью квадратных скобок:

```
L[…][…][…]…[…]
```

Какое исчисление выбрать: на доменах или кортежах? Не обязательно применять только одно исчисление. В каждой квадратной скобке мы можем выбирать любое, которое кажется нам удобным.

Продемонстрируем выбор, решая следующую задачу. Перенесемся с земли на небо – от задач, связанных с продажами и собаководством, перейдем к задаче о звездах, которая в 2025 году появилась в ЕГЭ.

Задача. Звезды заданы координатами на плоскости. Найти центроид – звезду, суммарное расстояние от которой до всех других звезд самое маленькое.

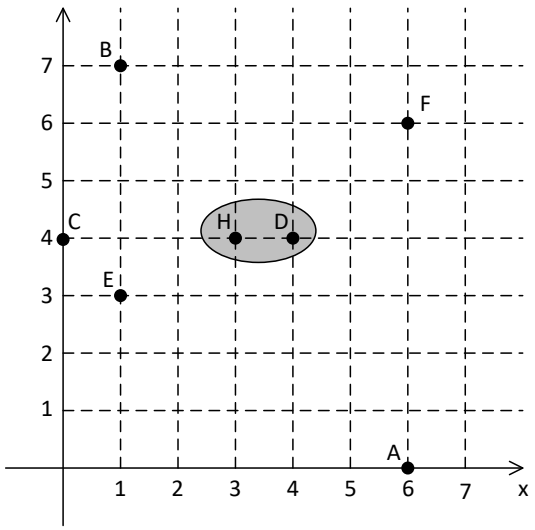
Мы решали задачи с точками на плоскости в главах раздела 2. Возьмем оттуда заготовку программы и напишем функцию вычисления расстояния между точками:

```
from decpy import var,queryclass

@queryclass
class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
    def __repr__(self):
        return "("+str(self.x)+","+str(self.y)+")"

def dist(A,B):
    return ((A.x-B.x)**2+(A.y-B.y)**2)**0.5

A=point(6,0)
B=point(1,7)
C=point(0,4)
D=point(4,4)
F=point(1,3)
G=point(6,6)
H=point(3,4)
print(point)
```

Результат:

```
{(4,4), (6,0), (1,7), (1,3), (0,4), (3,4), (6,6)}
```

Посмотрим на точки на плоскости:



Центроидом может быть одна из точек `D` или `H`.

Пошагово решим задачу.

Шаг 1. Поскольку нужно посчитать расстояния между всеми звездами, для начала нужно в запросе соединить каждую звезду с каждой, то есть сделать декартово произведение таблицы `point` самой с собой:

```
from decpy import var,queryclass

@queryclass
class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
    def __repr__(self):
        return "("+str(self.x)+","+str(self.y)+")"

def dist(A,B):
    return ((A.x-B.x)**2+(A.y-B.y)**2)**0.5

A=point(6,0)
B=point(1,7)
C=point(0,4)
D=point(4,4)
F=point(1,3)
G=point(6,6)
H=point(3,4)

L=point*point
for el in L:
    print(el)
```

Результат:

```
((6,0), (6,6))        ((3,4), (0,4))        ((0,4), (6,6))
((1,3), (0,4))        ((0,4), (3,4))        ((6,0), (6,0))
((6,0), (1,3))        ((0,4), (6,0))        ((6,6), (4,4))
((6,6), (1,7))        ((4,4), (6,0))        ((4,4), (6,6))
((6,6), (6,0))        ((1,7), (3,4))        ((6,0), (3,4))
((3,4), (4,4))        ((4,4), (3,4))        ((1,7), (1,3))
((6,6), (3,4))        ((1,7), (6,0))        ((4,4), (1,3))
((1,3), (1,7))        ((3,4), (1,7))        ((1,7), (6,6))
((0,4), (0,4))        ((6,6), (6,6))        ((1,3), (4,4))
((1,7), (0,4))        ((6,6), (1,3))        ((3,4), (1,3))
((4,4), (0,4))        ((6,0), (0,4))        ((3,4), (6,6))
((1,3), (6,0))        ((3,4), (3,4))        ((6,6), (0,4))
((0,4), (1,7))        ((1,3), (6,6))        ((0,4), (4,4))
((6,0), (4,4))        ((3,4), (6,0))        ((4,4), (4,4))
((1,3), (3,4))        ((1,3), (1,3))        ((1,7), (4,4))
((1,7), (1,7))        ((6,0), (1,7))
((4,4), (1,7))        ((0,4), (1,3))
```

Шаг 2. Для каждой пары звезд посчитаем расстояние:

```
from decpy import var,queryclass
...
```

```
a,b=var(2)
L=(point*point)[a,b,dist(a,b)]
for el in L:
    print(el)
```

Результат:

```
((0,4), (4,4), 4.0)
((1,7), (4,4), 4.242640687119285)
((3,4), (1,7), 3.605551275463989)
((6,0), (1,3), 5.830951894845301)
((0,4), (6,0), 7.211102550927978)
((0,4), (1,7), 3.1622776601683795)
((1,3), (6,0), 5.830951894845301)
((1,3), (4,4), 3.1622776601683795)
((4,4), (6,6), 2.8284271247461903)
((1,7), (6,0), 8.602325267042627)
((1,7), (6,6), 5.0990195135927845)
((0,4), (6,6), 6.324555320336759)
((6,0), (3,4), 5.0)
((6,0), (0,4), 7.211102550927978)
((3,4), (6,0), 5.0)
((6,6), (1,3), 5.830951894845301)
((6,6), (1,7), 5.0990195135927845)
((3,4), (4,4), 1.0)
((6,6), (4,4), 2.8284271247461903)
((6,6), (6,0), 6.0)
((6,6), (0,4), 6.324555320336759)
((0,4), (3,4), 3.0)
((3,4), (1,3), 2.23606797749979)
((1,3), (1,7), 4.0)
((6,0), (1,7), 8.602325267042627)
((4,4), (6,0), 4.47213595499958)
((0,4), (1,3), 1.4142135623730951)
((3,4), (0,4), 3.0)
((6,6), (3,4), 3.605551275463989)
((1,3), (6,6), 5.830951894845301)
((1,3), (0,4), 1.4142135623730951)
((6,0), (6,6), 6.0)
((4,4), (1,3), 3.1622776601683795)
((4,4), (0,4), 4.0)
((3,4), (6,6), 3.605551275463989)
((1,7), (3,4), 3.605551275463989)
((1,3), (3,4), 2.23606797749979)
((4,4), (3,4), 1.0)
((1,7), (1,3), 4.0)
((4,4), (1,7), 4.242640687119285)
((1,7), (0,4), 3.1622776601683795)
((6,0), (4,4), 4.47213595499958)
```

Шаг 3. Для каждой звезды составим список расстояний до всех других. Для этого сгруппируем строчки по первой (правой) звезде, а вторая (левая) становится ненужной:

```
from decpy import var,queryclass
...
a,b,d=var(3)
L=(point*point)[a,b,dist(a,b)][a.group(),None,d]
for el in L:
    print(el)
```

Результат:

```
((6,0), (8.602325267042627, 7.211102550927978,
4.47213595499958, 5.830951894845301, 5.0, 6.0))
((4,4), (4.47213595499958, 4.242640687119285, 1.0,
2.8284271247461903, 3.1622776601683795, 4.0))
((1,3), (1.4142135623730951, 5.830951894845301,
2.23606797749979, 3.1622776601683795, 4.0,
5.830951894845301))
((0,4), (3.0, 1.4142135623730951, 7.211102550927978,
3.1622776601683795, 4.0, 6.324555320336759))
((1,7), (5.0990195135927845, 8.602325267042627,
4.242640687119285, 3.605551275463989, 4.0,
3.1622776601683795))
((3,4), (3.605551275463989, 3.605551275463989,
2.23606797749979, 3.0, 5.0, 1.0))
((6,6), (6.0, 6.324555320336759, 5.830951894845301,
3.605551275463989, 2.8284271247461903, 5.0990195135927845))
```

Шаг 4. Для каждой звезды посчитаем сумму расстояний, сложив элементы списков:

```
from decpy import var,queryclass
...
a,b,d=var(3)
L=(point*point)[a,b,dist(a,b)][a.group(),None,d.sum()]
for el in L:
    print(el)
```

Результат:

```
((4,4), 19.705481427033433)
((0,4), 25.112149093806213)
((3,4), 18.44717052842777)
((1,7), 28.711814403387066)
((6,6), 29.688505128985025)
((1,3), 22.474462989731865)
((6,0), 37.116515667815484)
```

Заметим, что до сих пор мы писали программу в стиле исчисления на доменах. Дальше будет удобно писать в виде исчисления на кортежах.

Шаг 5. Найдем строчку с минимумом суммарного расстояния:

```
from decpy import var,queryclass
...
a,b,d,el=var(4)
L=(point*point)[a,b,dist(a,b)][a.group(),None,d.sum()].min(
el[1])
print(L)
```

Результат:

```
((3,4), 18.44717052842777)
```

Шаг 6. Наконец, суммарное расстояние нам не нужно, выведем только координаты звезды. То есть возьмем нулевой элемент коллекции. Приведем код программы полностью:

```
from decpy import var,queryclass

@queryclass
class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
    def __repr__(self):
        return "("+str(self.x)+","+str(self.y)+")"

def dist(A,B):
    return ((A.x-B.x)**2+(A.y-B.y)**2)**0.5

A=point(6,0)
B=point(1,7)
C=point(0,4)
D=point(4,4)
F=point(1,3)
G=point(6,6)
H=point(3,4)
a,b,d,el=var(4)
L=(point*point)[a,b,dist(a,b)][a.group(),None,d.sum()].min(
el[1])[0]
print(L)
```

Результат:

```
(3,4)
```

Центроидом оказалась точка H.

Для сравнения приведем еще три программы на Python, решающие эту же задачу, но в других стилях программирования.

Версия 2. Структурное программирование: циклы, накапливающаяся сумма, поиск минимума:

```
class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
    def __repr__(self):
        return "("+str(self.x)+","+str(self.y)+")"

def dist(A,B):
    return ((A.x-B.x)**2+(A.y-B.y)**2)**0.5

A=point(6,0)
B=point(1,7)
C=point(0,4)
D=point(4,4)
F=point(1,3)
G=point(6,6)
H=point(3,4)

L=[A,B,C,D,F,G,H]
m=10000000000000
z=None
for a in L:
    s=0
    for b in L:
        s=s+dist(a,b)
    if s<m:
        m=s
        z=a
print(z)
```

Версия 3. Структурное программирование, списки:

```
class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
    def __repr__(self):
        return "("+str(self.x)+","+str(self.y)+")"

def dist(A,B):
    return ((A.x-B.x)**2+(A.y-B.y)**2)**0.5

A=point(6,0)
```

```
B=point(1,7)
C=point(0,4)
D=point(4,4)
F=point(1,3)
G=point(6,6)
H=point(3,4)

L=[A,B,C,D,F,G,H]
S=[]
for a in L:
    D=[]
    for b in L:
        D.append(dist(a,b))
    S.append((sum(D),a))
print(min(S)[1])
```

Версия 4. Списочные выражения:

```
class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
    def __repr__(self):
        return "("+str(self.x)+","+str(self.y)+")"

def dist(A,B):
    return ((A.x-B.x)**2+(A.y-B.y)**2)**0.5

A=point(6,0)
B=point(1,7)
C=point(0,4)
D=point(4,4)
F=point(1,3)
G=point(6,6)
H=point(3,4)

L=[A,B,C,D,F,G,H]
print(min((sum(dist(a,b) for b in L),a) for a in L)[1])
```

Сравним теперь алгоритмические части программ:

Версия 1. Декларативное программирование с помощью библиотеки DecPy

```
a,b,d,el=var(4)
L=(point*point)[a,b,dist(a,b)][a.group(),None,d.sum()].
min(el[1])[0]
```

Версия 2. Структурное программирование: циклы, накапливающаяся сумма, поиск минимума:

```
m=10000000000000
z=None
for a in L:
    s=0
    for b in L:
        s=s+dist(a,b)
    if s<m:
        m=s
        z=a
print(z)
```

Версия 3. Структурное программирование, списки:

```
S=[]
for a in L:
    D=[]
    for b in L:
        D.append(dist(a,b))
    S.append((sum(D),a))
print(min(S)[1])
```

Версия 4. Списочные включения:

```
print(min((sum(dist(a,b) for b in L),a) for a in L)[1])
```

Версия 2 является классическим решением. Так писали бы программу на Fortran, Basic, Pascal и языках семейства C++. Удобные коллекции (списки, множества, словари), встроенные в Python, подталкивают программиста к версии 3, которую можно считать переходным этапом к версии 4. Списочные включения экономны и позволяют написать алгоритм в одну строку. Их можно считать разновидностью декларативного программирования, «родного» для Python. Программа декларативного программирования с помощью библиотеки DecPy немного больше, но понятнее.

Выводы

Как показала задача поиска центроида скопления звезд, можно добиться простоты кода, умело комбинируя запросы в стилях исчислений на кортежах и на доменах. Но конкуренцию такому коду составляют списочные включения Python, в которых циклы встроены внутри объявлений списков (версия 4 последней программы). Можно поспорить, какая версия лучше – первая или четвертая. Но для задач следующего раздела преимущество библиотеки DecPy будет неоспоримо. Пока же можно считать декларативное программирование с помощью библиотеки DecPy однострочником без циклов. Циклы библиотека DecPy создает сама.

4. Запросы в стиле исчисления предикатов первого порядка

Закончив приводить примеры запросов в стиле исчисления на кортежах (раздел 2) и исчисления на доменах (раздел 3), пора перейти к запросам в стиле ***исчисления предикатов первого порядка*** – третьему направлению декларативного программирования.

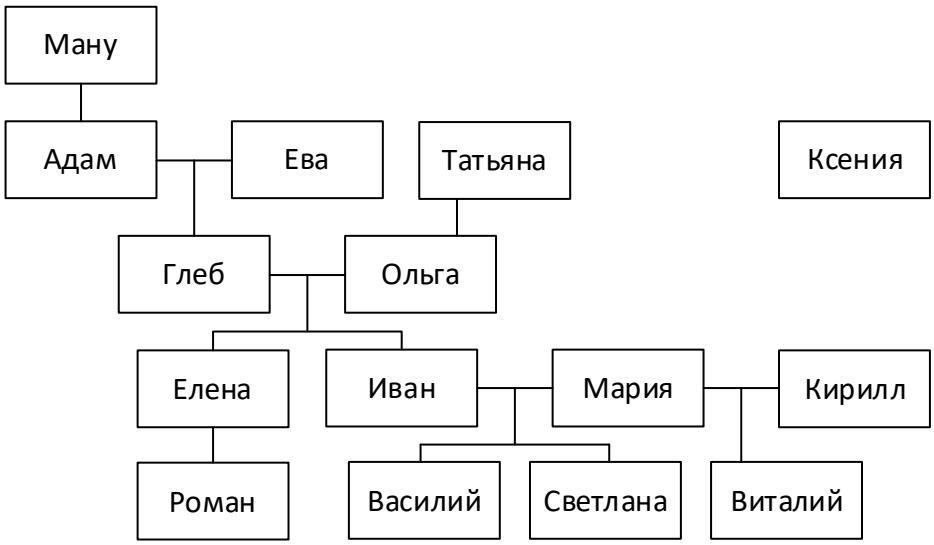
На исчислении предикатов первого порядка основывается стиль ***логическое программирование***, ярким представителем которого является язык искусственного интеллекта Prolog, на котором разрабатывают экспертные системы.

Логическое программирование долгое время стояло особняком в мире IT-технологий, и в нем сложилась своя терминология: факты, предикаты и т.д. Чтобы не вносить путаницу, мы будем пользоваться терминологией из предыдущих глав, ведь каждому термину из логического программирования можно найти аналог если не в программировании общего назначения, то в базах данных.

4.1. Родословное древо – постановка задачи

Будем решать задачи про родословное древо. Задачи будут состоять в поиске людей, находящихся в родственных отношениях, например, поиске братьев, предков, родственников и т.п.

В примерах запросы будем делать к следующему родословному древу:



В нем Ксения не входит в родословное древо и приведена для контроля правильности запросов.

На этот раз представим людей в программе как экземпляры класса «Человек». Родословное древо будет представлять собой ***рекурсивную структуру данных*** (структуру, узел которой содержит ссылки на узлы того же самого типа). В нашем случае выберем базовые родственные отношения «мать» и «отец». У каждого экземпляра класса «Человек» будем хранить помимо простых атрибутов «Имя» и «Год рождения» еще два атрибута «Мать» и «Отец». Причем «Мать» и «Отец» также будут экземплярами класса «Человек». Сразу оформим класс «Человек» декоратором `queryclass`, который сохраняет экземпляры класса во множество с таким же именем, как у класса, и позволяет писать к нему запросы (см. главу 2.6.):

```
from decpy import var, queryclass

@queryclass
class person:
    def __init__(self,name,age,mother=None,father=None):
        self.name=name
        self.age=age
        self.mother=mother
        self.father=father
    def __repr__(self):
        return self.name+" "+str(self.age)

p0=person("Ману",1931)
p1=person("Адам",1950,None,p0)
p2=person("Ева",1951)
p3=person("Татьяна",1952)
p4=person("Глеб",1970,p2,p1)
p5=person("Ольга",1970,p3)
p6=person("Иван",1990,p5,p4)
p7=person("Мария",1991)
p8=person("Василий",2000,p7,p6)
p9=person("Светлана",2012,p7,p6)
p10=person("Елена",2012,p5,p4)
p11=person("Роман",2024,p10)
p12=person("Кирилл",2012)
p13=person("Виталий",2024,p7,p12)
p14=person("Ксения",2024)

print(person)
```

Результат:

```
{Татьяна 1952, Адам 1950 Ману, Ману 1931, Василий 2000
Мария Иван, Кирилл 2012, Глеб 1970 Ева Адам, Мария 1991,
Виталий 2024 Мария Кирилл, Ева 1951, Елена 2012 Ольга Глеб,
Иван 1990 Ольга Глеб, Светлана 2012 Мария Иван, Ольга 1970
Татьяна, Роман 2024 Елена, Ксения 2024}
```

4.2. Первые понятия

Узнать, кто является матерью Василия, достаточно просто, ведь у экземпляра класса «Человек» есть атрибут «Мать»:

```
print(p8.mother)
```

Но кто является детьми у Марии, так просто не узнать, надо перебирать всё родословное древо. Поручим эту работу библиотеке DecPy.

Понятие №1. **Мать**. Нужно создать таблицу (в терминах логического программирования – двуместный предикат), в которой в первой колонке будет ребенок, а во второй – мать.

```
from decpy import var, queryclass

@queryclass
class person:
    def __init__(self,name,age,mother=None,father=None):
        self.name=name
        self.age=age
        self.mother=mother
        self.father=father
    def __repr__(self):
        return self.name+" "+str(self.age)


p0=person("Ману",1931)
p1=person("Адам",1950,None,p0)
p2=person("Ева",1951)
p3=person("Татьяна",1952)
p4=person("Глеб",1970,p2,p1)
p5=person("Ольга",1970,p3)
p6=person("Иван",1990,p5,p4)
p7=person("Мария",1991)
p8=person("Василий",2000,p7,p6)
p9=person("Светлана",2012,p7,p6)
p10=person("Елена",2012,p5,p4)
p11=person("Роман",2024,p10)
p12=person("Кирилл",2012)
p13=person("Виталий",2024,p7,p12)
p14=person("Ксения",2024)
```

```
X,Y,el=var(3)
mother = (person*person)[X,Y,X.mother==Y]
for p in mother:
    print(p)
```

Результат:

```
(Виталий 2024, Мария 1991)
(Светлана 2012, Мария 1991)
(Ольга 1970, Татьяна 1952)
(Елена 2012, Ольга 1970)
(Роман 2024, Елена 2012)
(Василий 2000, Мария 1991)
(Иван 1990, Ольга 1970)
(Глеб 1970, Ева 1951)
```

Теперь легко найти детей Марии, написав запрос к понятию «Мать»:

```
from decpy import var, queryclass
...
X,Y,el=var(3)
mother = (person*person)[X,Y,X.mother==Y]
print(mother[X,p7])
```

Результат:

```
{(Виталий 2024, Мария 1991), (Светлана 2012, Мария 1991),
(Василий 2000, Мария 1991)}
```

Если не нужно упоминать Марию в результате запроса, то сделаем добавку:

```
print(mother[X,p7][X,None])
```

Результат:

```
{Василий 2000, Виталий 2024, Светлана 2012}
```

Аналогично понятию «Мать» введем понятие «Отец»:

```
father = (person*person)[X,Y,X.father==Y]
```

Понятие 2. Родитель – это мать или отец. Объединим результаты двух понятий:

```
from decpy import var, queryclass
...
X,Y=var(2)
mother = (person*person)[X,Y,X.mother==Y]
father = (person*person)[X,Y,X.father==Y]
parent = mother | father
for p in parent:
    print(p)
```

Результат:

```
(Виталий 2024, Мария 1991)
(Роман 2024, Елена 2012)
(Василий 2000, Иван 1990)
(Иван 1990, Глеб 1970)
(Глеб 1970, Ева 1951)
(Елена 2012, Глеб 1970)
(Василий 2000, Мария 1991)
(Елена 2012, Ольга 1970)
(Иван 1990, Ольга 1970)
(Виталий 2024, Кирилл 2012)
(Светлана 2012, Иван 1990)
(Светлана 2012, Мария 1991)
(Адам 1950, Ману 1931)
(Ольга 1970, Татьяна 1952)
(Глеб 1970, Адам 1950)
```

Заметим, что в этом разделе вместо слова «запрос», как в предыдущих разделах, мы используем слово «понятие». Ведь их результат – это не просто выборка данных. Он несет в себе определенную семантику. На основе некоторых понятий будут строиться другие. И «ленивые запросы» в случае понятий вполне оправданы. Определение матери остается неизменным при пополнении родословного древа, список же матерей будет обновляться.

4.3. Ограничения и хороший стиль

Найдем для Ивана ближайших родственников – его родителей и детей. Прежде чем формулировать понятие «ближайший родственник», надо сформулировать понятие ребенок – фактически поменять местами колонки в таблице «родитель».

Понятие 3. Ребенок:

```
from decpy import var, queryclass
...
mother = (person*person)[X,Y,X.mother==Y]
father = (person*person)[X,Y,X.father==Y]
parent = mother | father
child = var()
child[X,Y] |= parent[Y,X]
for p in child:
    print(p)
```

Результат:

```
(Мария 1991, Василий 2000)
(Елена 2012, Роман 2024)
(Адам 1950, Глеб 1970)
(Глеб 1970, Иван 1990)
(Мария 1991, Светлана 2012)
(Кирилл 2012, Виталий 2024)
(Ману 1931, Адам 1950)
(Глеб 1970, Елена 2012)
(Иван 1990, Светлана 2012)
(Татьяна 1952, Ольга 1970)
(Иван 1990, Василий 2000)
(Ольга 1970, Иван 1990)
(Мария 1991, Виталий 2024)
(Ева 1951, Глеб 1970)
(Ольга 1970, Елена 2012)
```

Обмен местами колонок интуитивно понятен, но обратите внимание, что мы использовали другой оператор – «|=». В Python он значит «добавить к множеству дополнительные элементы». Получается, что строчкой

```
child = var()
```

мы создали пустое множество, а строчкой

```
child[X,Y] |= parent[Y,X]
```

дописали к нему элементы.

Но в библиотеке DecPy оператору «|=» придается другой смысл – «равно по определению». С помощью этого оператора мы даем определения понятий. На языке Prolog оператору «|=» соответствует похожий ему по написанию оператор «:-». Перепишем ранее введенные понятия с помощью оператора «|=»:

```
from decpy import var, queryclass,table
...
X,Y,mother,father,parent,child=var(6)
mother |= (person*person)[X,Y,X.mother==Y]
father |= (person*person)[X,Y,X.father==Y]
parent |= mother | father
child[X,Y] |= parent[Y,X]
for p in child:
    print(p)
```

Проверим, работает ли оператор «|=» в смысле Python – дописывает ли он новые элементы к множеству? Для этого разобьем определение родителя на две строчки:

```
...
parent |= mother
parent |= father
...
```

Результат:

```
(Глеб 1970, Иван 1990)
(Ману 1931, Адам 1950)
(Кирилл 2012, Виталий 2024)
(Иван 1990, Василий 2000)
(Иван 1990, Светлана 2012)
(Глеб 1970, Елена 2012)
(Адам 1950, Глеб 1970)
```

Видно, что дети теперь вывелись только у отцов. Вторая строчка определения родителя отменила первую, в результате матери забыты!

Но источник вдохновения библиотеки DecPy – язык Prolog имеет возможность множественных определений! Пока это является ограничением объекта `var` библиотеки DecPy (возможно, оно будет преодолено в будущем). Но такого ограничения нет у объекта `table`! У `table` оператор «`|=`» работает как в смысле Python – дописывает к множеству элементы, так и в смысле Prolog – формулирует новые понятия.

Посмотрим на строчку:

```
X,Y,mother,father,parent,child=var(6)
```

Она плоха со стилистической точки зрения: внутренние вспомогательные переменные `X` и `Y` смешались с понятиями `mother`, `father`, `parent`, `child`. Объявим их по отдельности, причем понятия объявим как `table`:

```
X,Y=var(2)
mother,father,parent,child=table(4)
```

Проверим работу:

```
from decpy import var, queryclass,table
...
X,Y=var(2)
mother,father,parent,child=table(4)
mother |= (person*person)[X,Y,X.mother==Y]
father |= (person*person)[X,Y,X.father==Y]
parent |= mother
parent |= father
child[X,Y] |= parent[Y,X]
for p in child:
    print(p)
```

Результат:

```
(Глеб 1970, Иван 1990)
(Елена 2012, Роман 2024)
(Ольга 1970, Елена 2012)
(Мария 1991, Виталий 2024)
(Иван 1990, Василий 2000)
(Татьяна 1952, Ольга 1970)
(Мария 1991, Светлана 2012)
(Глеб 1970, Елена 2012)
(Иван 1990, Светлана 2012)
(Мария 1991, Василий 2000)
(Ману 1931, Адам 1950)
(Адам 1950, Глеб 1970)
(Кирилл 2012, Виталий 2024)
(Ева 1951, Глеб 1970)
(Ольга 1970, Иван 1990)
```

Как видно, теперь матери добавились к своим детям! Все работает правильно.

Понятие 4. Ближайший родственник – родитель или ребенок.

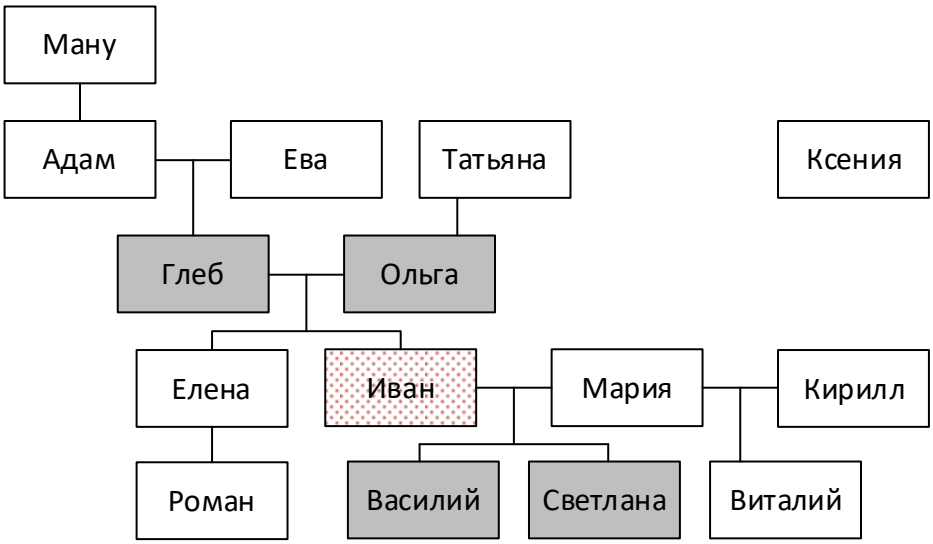
Напишем понятие «Ближайший родственник» и выведем ближайших родственников Ивана:

```
from decpy import var, queryclass, table
...
X,Y=var(2)
mother,father,parent,child,near=table(5)
mother |= (person*person)[X,Y,X.mother==Y]
father |= (person*person)[X,Y,X.father==Y]
parent |= mother | father
child[X,Y] |= parent[Y,X]
near |= parent | child
print(near[p6,X][None,X])
```

Результат:

```
{Василий 2000, Глеб 1970, Ольга 1970, Светлана 2012}
```

Проверим ближайших родственников Ивана в родословном древе:



Раз мы заговорили о хорошем стиле, то переместим определения понятий в коде программы, чтобы они стояли выше данных. Ведь определения понятий не изменятся, если изменятся данные. Заодно убедимся, что они работают как ленивые вычисления. Приведем код программы полностью:

```
from decpy import var, queryclass, table

@queryclass
class person:
    def __init__(self,name,age,mother=None,father=None):
        self.name=name
        self.age=age
        self.mother=mother
        self.father=father
    def __repr__(self):
        return self.name+" "+str(self.age)

X,Y=var(2)
mother,father,parent,child,near=table(5)
mother |= (person*person)[X,Y,X.mother==Y]
father |= (person*person)[X,Y,X.father==Y]
parent |= mother | father
child[X,Y] |= parent[Y,X]
near |= parent | child

p0=person("Ману",1931)
p1=person("Адам",1950,None,p0)
p2=person("Ева",1951)
p3=person("Татьяна",1952)
p4=person("Глеб",1970,p2,p1)
p5=person("Ольга",1970,p3)
p6=person("Иван",1990,p5,p4)
p7=person("Мария",1991)
p8=person("Василий",2000,p7,p6)
```

```
p9=person("Светлана",2012,p7,p6)
p10=person("Елена",2012,p5,p4)
p11=person("Роман",2024,p10)
p12=person("Кирилл",2012)
p13=person("Виталий",2024,p7,p12)
p14=person("Ксения",2024)

print(near[p6,X][None,X])
```

Теперь код корректен и стилистически выдержан (правда, далее будет еще одно улучшение стиля). Вначале идет описание структуры исходных данных в виде декорированного класса «Человек». Затем идут определения вторичных понятий «мать», «родитель» и т.д. Причем понятия и переменные не смешиваются. Понятия объявлены как `table`, переменные как `var`. Потом идут сами данные родословного древа и в самом конце – запрос (вывести ближайших родственников конкретного человека). Будем и дальше следовать такой структуре программы.

4.4. Еще немного простых понятий

Понятие 5. Бабушка и дедушка. Создадим таблицы-пары: «внук (внучка) / бабушка (дедушка)».

Дадим определение на основе имеющихся у нас понятий. Бабушка(дедушка) – это родитель родителя. То есть таблица «Родитель» входит в определение два раза. Значит, нам нужно сделать декартово произведение `parent` с самим собой.

```
...
X,Y=var(2)
mother,father,parent,grandparent = table(4)
...
grandparent = parent*parent
...
print(grandparent)
```

Результат получился очень большой. Для примера выведем только два элемента:

```
((Светлана 2012, Иван 1990), (Иван 1990, Ольга 1970)),
((Адам 1950, Ману 1931), (Василий 2000, Иван 1990)),...
```

Видно, что первая пара позволяет найти бабушку Светланы – Ольгу (отец у Светланы – Иван, и он же – сын Ольги), а вторая – нет. Заменим оператор «*» на «**», чтобы убрать внутреннюю структуру, и сделаем отбор по совпадению промежуточного человека:

```
from decpy import var, queryclass, table
...
X,Y,Z=var(3)
...
grandparent |= (parent**parent)[X,Z,Z,Y]

p0=person("Ману",1931)
...
p14=person("Ксения",2024)

for p in grandparent:
    print(p)
```

Результат:

```
(Елена 2012, Глеб 1970, Адам 1950)
(Василий 2000, Иван 1990, Ольга 1970)
(Глеб 1970, Адам 1950, Ману 1931)
(Иван 1990, Ольга 1970, Татьяна 1952)
(Василий 2000, Иван 1990, Глеб 1970)
(Иван 1990, Глеб 1970, Адам 1950)
(Иван 1990, Глеб 1970, Ева 1951)
(Светлана 2012, Иван 1990, Ольга 1970)
(Елена 2012, Ольга 1970, Татьяна 1952)
(Светлана 2012, Иван 1990, Глеб 1970)
(Роман 2024, Елена 2012, Ольга 1970)
(Роман 2024, Елена 2012, Глеб 1970)
(Елена 2012, Глеб 1970, Ева 1951)
```

Мы получили связь внуков и бабушек/дедушек, но в ответе еще присутствует промежуточное звено (сын/дочь бабушки/дедушки и он же родитель внука/внучки). Избавиться от него можно, введя дополнительные квадратные скобки:

```
from decpy import var, queryclass, table
...
X,Y,Z=var(3)
...
grandparent |= (parent**parent)[X,Z,Z,Y][X,None,Y]

p0=person("Ману",1931)
...
p14=person("Ксения",2024)

for p in grandparent:
    print(p)
```

Результат верен, но код далек от идеала. У нас вряд ли получилось бы без ошибок написать все определение сразу:

```
grandparent |= (parent**parent)[X,Z,Z,Y][X,None,Y]
```

Пока это определение написано в стиле исчисления на доменах. Но я уже упоминал исчисление предикатов первого порядка. Они родственны друг другу, но ***логическое программирование*** (основанное на исчислении предикатов первого порядка) дает новые языковые возможности.

Первой возможностью является то, что мы можем перенести правую квадратную скобку в левую часть равенства, написав там без `None`, какие атрибуты мы хотим видеть в результате:

```
grandparent[X,Y] |= (parent**parent)[X,Z,Z,Y]~~[X,None,Y]~~
```

В декартовом произведении могут участвовать более двух таблиц. Тогда в веренице переменных можно по невнимательности сделать ошибку. Библиотека DecPy, как и Prolog, позволяет поставить переменные рядом с обрабатываемыми таблицами:

```
grandparent[X,Y] |= parent[X,Z]**parent[Z,Y]~~[X,Z,Z,Y]~~
```

Это определение:

```
grandparent[X,Y] |= parent[X,Z]**parent[Z,Y]
```

намного проще, чем в стиле исчисления на доменах:

```
grandparent |= (parent**parent)[X,Z,Z,Y][X,None,Y]
```

И его можно понять в словесной форме: `Y` является дедушкой (бабушкой) `X`, если существует человек `Z`, такой, что `Z` является одновременно родителем `X` и ребенком для `Y`.

Напомню ход рассуждений:

```
grandparent = parent*parent
grandparent = parent**parent
grandparent |= (parent**parent)[X,Z,Z,Y]
grandparent |= (parent**parent)[X,Z,Z,Y][X,None,Y]
grandparent[X,Y] |= (parent**parent)[X,Z,Z,Y]
grandparent[X,Y] |= parent[X,Z]**parent[Z,Y]
```

Последнее определение проще, чем промежуточные определения посередине хода рассуждений! `table` подталкивает нас к тому, чтобы мы формулировали понятия сразу в стиле логического программирования. Хорошим стилем является явное указание в левой части в квадратных скобках переменных – атрибутов результатов, даже если это не влияет на результат.

Кроме того, нет необходимости в создании реверсивных понятий родитель – ребенок, дедушка – внук, так как если в итоговой таблице более двух атрибутов, создавать все комбинации бессмысленно.

Перепишем наши определения, убрав понятие `child` и добавив квадратные скобки в левой части определения понятий. Приведем программу полностью:

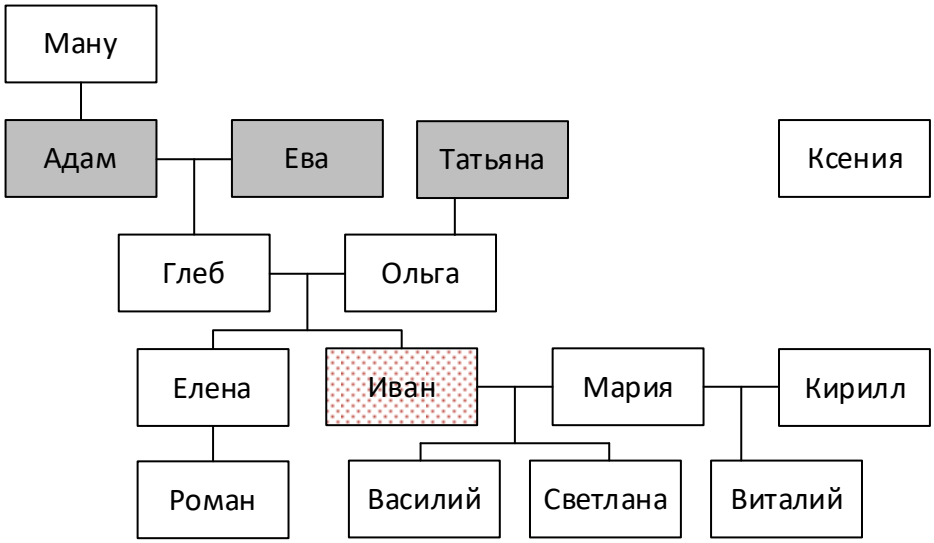
```
from decpy import var, queryclass, table

@queryclass
class person:
    def __init__(self,name,age,mother=None,father=None):
        self.name=name
        self.age=age
        self.mother=mother
        self.father=father
    def __repr__(self):
        return self.name+" "+str(self.age)

X,Y,Z=var(3)
mother,father,parent,child,near,grandparent=table(6)
mother[X,Y] |= (person*person)[X,Y,X.mother==Y]
father[X,Y] |= (person*person)[X,Y,X.father==Y]
parent[X,Y] |= mother[X,Y] | father[X,Y]
near[X,Y] |= parent[X,Y] | parent[Y,X]
grandparent[X,Y] |= parent[X,Z]**parent[Z,Y]

p0=person("Ману",1931)
p1=person("Адам",1950,None,p0)
p2=person("Ева",1951)
p3=person("Татьяна",1952)
p4=person("Глеб",1970,p2,p1)
p5=person("Ольга",1970,p3)
p6=person("Иван",1990,p5,p4)
p7=person("Мария",1991)
p8=person("Василий",2000,p7,p6)
p9=person("Светлана",2012,p7,p6)
p10=person("Елена",2012,p5,p4)
p11=person("Роман",2024,p10)
p12=person("Кирилл",2012)
p13=person("Виталий",2024,p7,p12)
p14=person("Ксения",2024)

print("Ближайшие родственники Ивана:")
print(near[p6,X][None,X])
print("Бабушки и дедушки Ивана:")
print(grandparent[p6,X][None,X])
```

Результат:

```
Ближайшие родственники Ивана:
{Глеб 1970, Ольга 1970, Светлана 2012, Василий 2000}
```

```
Бабушки и дедушки Ивана:
{Адам 1950, Ева 1951, Татьяна 1952}
```

Проверим бабушек и дедушек Ивана на родословном древе:



Теперь программа полностью соответствует хорошему стилю и написана в парадигме логического программирования.

Понятие 6. Брат и сестра

Дадим определение братьям и сестрам сразу в стиле логического программирования. Люди являются братьями и сестрами, если у них есть общий родитель. Более строгое определение через переменные: `X` является братом (сестрой) для `Y`, если есть человек `Z`, такой, что `Z` является родителем и для `X`, и для `Y`. Запрограммируем это определение:

```
...
brother[X,Y] |= parent[X,Z]**parent[Y,Z]
...
print("Братья и сестры Василия:")
print(brother[p8,X][None,X])
```

Результат:

```
Братья и сестры Василия:
{Василий 2000, Виталий 2024, Светлана 2012}
```

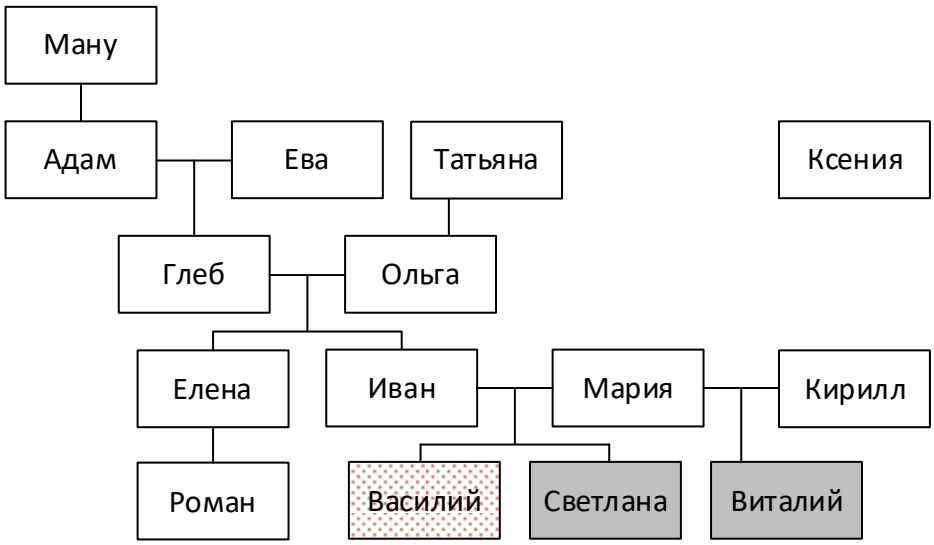
Если внимательно посмотреть на результат, то мы увидим, что Василий является братом самому себе. Нужно указать, что `X` не равен `Y`. Сделать это можно в левых скобках или последних скобках:

```
brother[X,Y,X!=Y] |= parent[X,Z]**parent[Y,Z]
brother[X,Y] |= parent[X,Z]**parent[Y,Z,X!=Y]
```

Результат:

```
{Виталий 2024, Светлана 2012}
```

Посмотрим братьев и сестер Василия на родословном древе:



Заметим, что по нашему определению братом Василию является также единоутробный брат Виталий (у них мать одна, а отцы - разные).

Можно дать строгое определение для родного брата/сестры. Это люди, у которых общие мать и отец:

```
rodbrat[X,Y,X!=Y] |=
mother[X,Z]**mother[Y,Z]**father[X,U]**father[Y,U]
```

Результат:

```
Родные братья и сестры Василия:
{Светлана 2012}
```

4.5. Рекурсивные определения

Понятие 7. Предок. К предкам относятся родители, бабушки и дедушки, прабабушки и прадедушки и т.д. по всей прямой линии родства.

Если мы попытаемся решить задачу перечислением:

```
ancestor[X,Y] |= parent[X,Y] | grandparent[X,Y] | ...
```

то нам придется вводить понятие прадедушки, прапрадедушки и т.д. И все равно мы не сможем таким образом написать понятие для родословного древа любой высоты. Нужно общее решение.

Попробуем сформулировать общее решение через переменные. `Y` является предком для `X`, если `Y` является родителем для `X` или… Что делать, если `Y` не является родителем `X`? Это значит, что у `X` есть другой родитель – `Z`. А кем

приходится Y для Z? Ну конечно же предком! Запрограммируем это определение:

```
ancestor[X,Y] |= parent[X,Y] | parent[X,Z]**ancestor[Z,Y]
```

Получается, что мы составили определение предка через самого себя! Неужели это будет работать? Такое определение называется рекурсивным. Часть без рекурсии называется терминальным случаем. У нас это (выделенно жирным):

```
ancestor[X,Y] |= parent[X,Y] | parent[X,Z]**ancestor[Z,Y]
```

А следующая ветка (выделена жирным):

```
ancestor[X,Y] |= parent[X,Y] | parent[X,Z]**ancestor[Z,Y]
```

является рекурсивной веткой.

Рекурсии являются сейчас важным приемом программирования, и их поддерживают все популярные языки программирования. Впервые с похожими записями начинающие программисты встречаются, когда видят рекуррентные формулы вида:

```
s=s+a
```

Но не получится ли так, что определение сработает как рекуррентная формула, то есть в предки попадут только родители с бабушками и дедушками. А затем надо будет повторять определение, чтобы добавились прабабушки и прадедушки? Проверим. В программе оставим только необходимые для определения предка понятия:

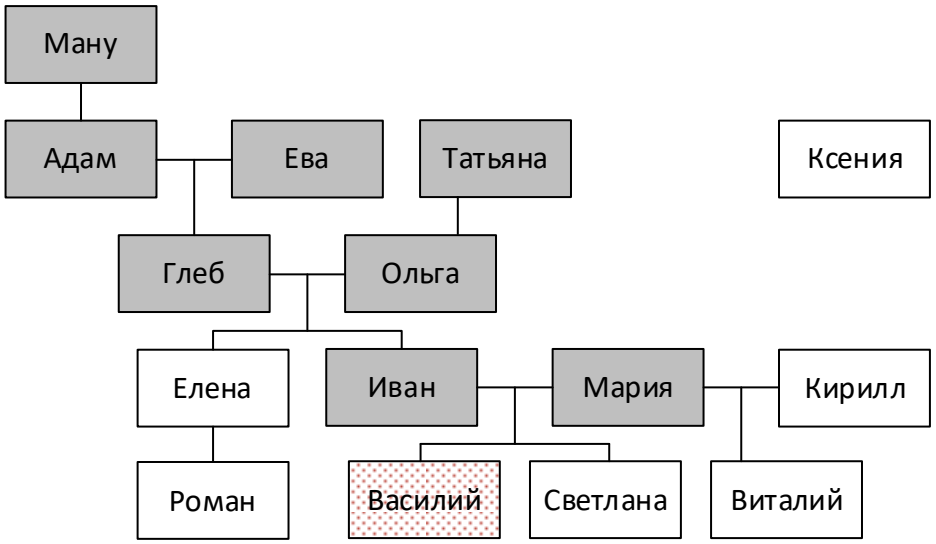
```
from decpy import var, queryclass, table

@queryclass
class person:
    def __init__(self,name,age,mother=None,father=None):
        self.name=name
        self.age=age
        self.mother=mother
        self.father=father
    def __repr__(self):
        return self.name+" "+str(self.age)

X,Y,Z=var(3)
mother,father,parent,ancestor=table(4)
mother[X,Y] |= (person*person)[X,Y,X.mother==Y]
father[X,Y] |= (person*person)[X,Y,X.father==Y]
parent[X,Y] |= mother[X,Y] | father[X,Y]
ancestor[X,Y] |= parent[X,Y] | parent[X,Z]**ancestor[Z,Y]
```

```
p0=person("Ману",1931)
...
p14=person("Ксения",2024)

print("Предки Василия:")
print(ancestor[p8,X][None,X])
```

Результат:

```
Предки Василия:
{Иван 1990, Адам 1950, Татьяна 1952, Глеб 1970, Ману 1931,
Ева 1951, Ольга 1970, Мария 1991}
```

Проверим предков Василия на родословном древе:



Видно, что по определению нашлись все предки Василия по прямой линии. Реализация рекурсии в декларативных языках является большой проблемой и мало где делается так изящно, как в Prolog, а теперь еще и в библиотеке DecPy.

Продолжим формулировать понятия для родословного древа.

Понятие 8. Кровные родственники. Это люди с общими генами, то есть прямые предки и потомки, а также люди, которые предками и потомками друг другу не являются, но у них есть общий предок. Заметим, что жена кровным родственником не является.

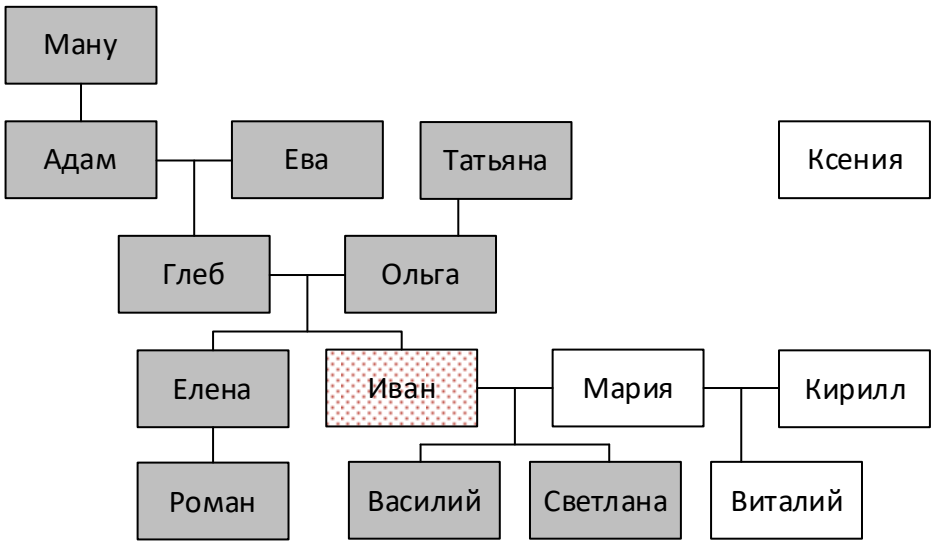
Запрограммируем это понятие и найдем родственников Ивана:

```
from decpy import var, queryclass, table

@queryclass
class person:
    def __init__(self,name,age,mother=None,father=None):
        self.name=name
        self.age=age
        self.mother=mother
        self.father=father
    def __repr__(self):
        return self.name+" "+str(self.age)

X,Y,Z=var(3)
mother,father,parent,ancestor,rod=table(5)
mother[X,Y] |= (person*person)[X,Y,X.mother==Y]
father[X,Y] |= (person*person)[X,Y,X.father==Y]
parent[X,Y] |= mother[X,Y] | father[X,Y]
ancestor[X,Y] |= parent[X,Y] | parent[X,Z]**ancestor[Z,Y]
rod[X,Y,X!=Y] |= ancestor[X,Y] | ancestor[Y,X] |
ancestor[X,Z]**ancestor[Y,Z]

p0=person("Ману",1931)
...
p14=person("Ксения",2024)
print("Родственники Ивана:")
print(rod[p6,X][None,X])
```

Результат:

```
Родственники Ивана:
{Елена 2012, Адам 1950, Василий 2000, Светлана 2012, Ольга
1970, Глеб 1970, Ману 1931, Роман 2024, Ева 1951, Татьяна
1952}
```

Проверим родственников Ивана на родословном древе:



Понятие 9. Свояк

В русской системе родства есть много понятий для близких людей, которые кровными родственниками не являются, например, жены и мужья, шурины, невестки и золовки, теща и свекровь и т.д. Назовем их свояками. Это люди, которых связывает либо общий родственник (например, мужа и жену связывают дети), либо цепочка общих родственников.

Определение это рекурсивно и строится точно так же, как и определение предка, только вместо родителя в определении будет родственник:

```
svoy[X,Y,X!=Y] |= rod[X,Y] | rod[X,Z]**svoy[Z,Y]
```

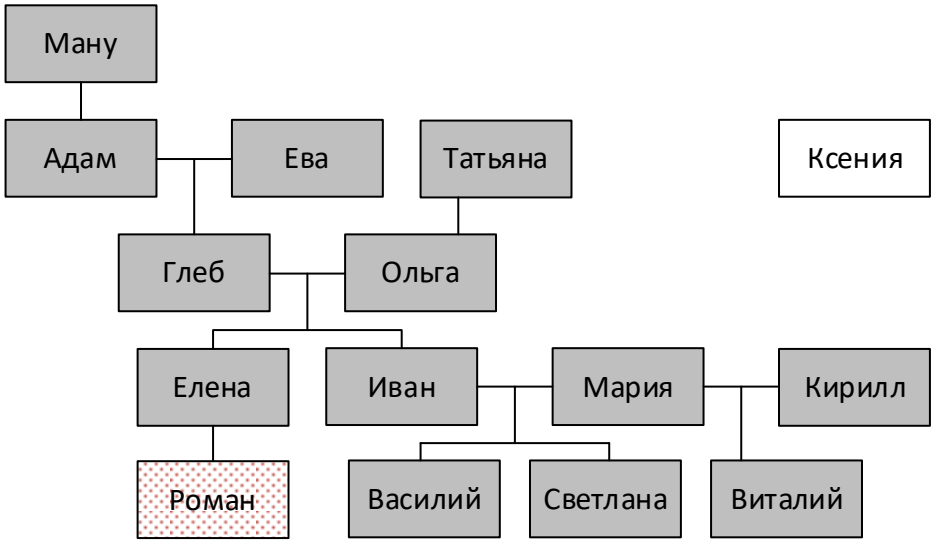
Если мы хотим исключить из свояков кровных родственников, то сделаем это с помощью вычитания множеств:

```
from decpy import var, queryclass, table

@queryclass
class person:
    def __init__(self,name,age,mother=None,father=None):
        self.name=name
        self.age=age
        self.mother=mother
        self.father=father
    def __repr__(self):
        return self.name+" "+str(self.age)

X,Y,Z=var(3)
mother,father,parent,ancestor,rod,svoy=table(6)
mother[X,Y] |= (person*person)[X,Y,X.mother==Y]
father[X,Y] |= (person*person)[X,Y,X.father==Y]
parent[X,Y] |= mother[X,Y] | father[X,Y]
ancestor[X,Y] |= parent[X,Y] | parent[X,Z]**ancestor[Z,Y]
rod[X,Y,X!=Y] |= ancestor[X,Y] | ancestor[Y,X] |
ancestor[X,Z]**ancestor[Y,Z]
svoy[X,Y,X!=Y] |= rod[X,Y] | rod[X,Z]**svoy[Z,Y]

p0=person("Ману",1931)
...
p14=person("Ксения",2024)

print("Свояки Романа:")
print(svoy[p11,X][None,X])
print("Свояки Романа без кровных родственников:")
print(svoy[p11,X][None,X]-rod[p11,X][None,X])
```

Результат:

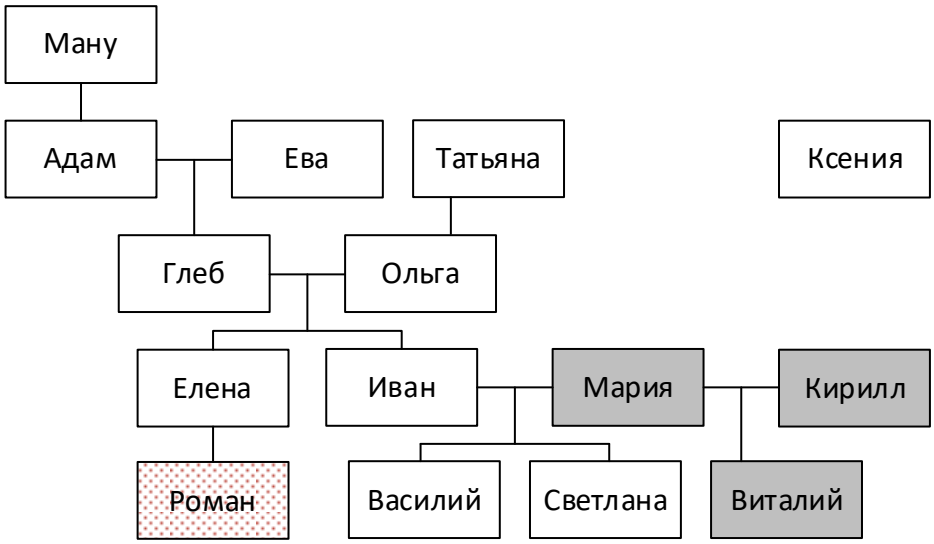
```
Свояки Романа:
{Адам 1950, Василий 2000, Кирилл 2012, Мария 1991, Глеб
1970, Ману 1931, Светлана 2012, Виталий 2024, Татьяна 1952,
Иван 1990, Ева 1951, Ольга 1970, Елена 2012}
Свояки Романа без кровных родственников:
{Мария 1991, Виталий 2024, Кирилл 2012}
```

Посмотрим свояков Романа с родственниками на родословном древе:



Из анализа результата видно, что в свояки с родственниками попали все люди из родословного древа, кто хоть как-то связан с Романом. Отсутствует только Ксения, которая не соединена с нашим деревом никакой связью.

Посмотрим на свояков Романа без кровных родственников на родословном древе:

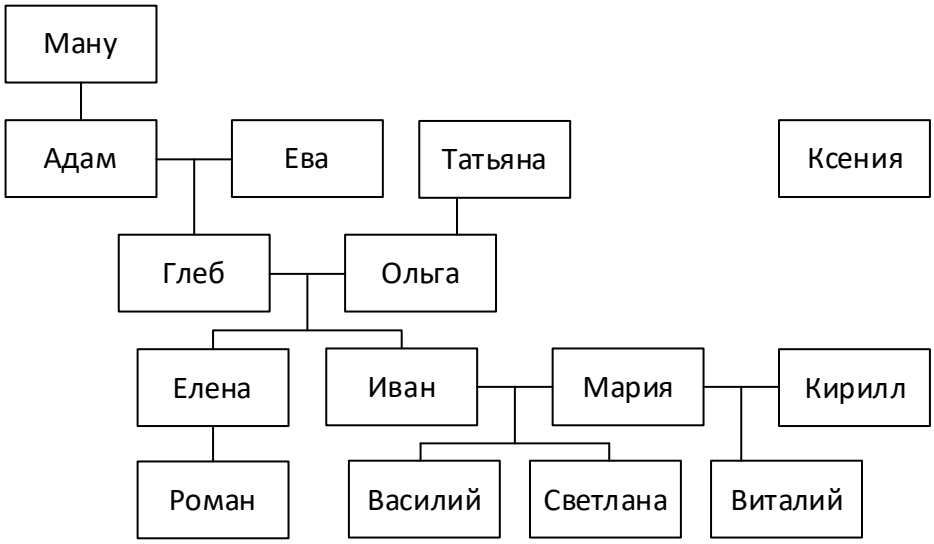


Замечу, что понятие свояка мне так и не удалось запрограммировать на Prolog. Компилятор уходил в бесконечный цикл. Prolog содержит оператор для

прерывания поиска, но использование этого оператора вредит чистоте стиля и ухудшает восприятие программы. Библиотека DecPy содержит проверки зацикливания и прерывает бесконечные циклы сама.

Понятие 10. Поколение.

Посмотрим еще раз на родословное древо:



Для красоты воприятия я расположил людей по уровням – поколениям. Интуитивно понятно, какие люди входят в одно поколение. Но тяжело дать этому понятию формальное определение.

Заметим, что люди находятся в одном поколении, если они являются братьями / сестрами (у них есть общий родитель) или мужем и женой (у них есть общие дети):

```
generation[X,Y] |=
parent[X,Z]**parent[Y,Z]|parent[Z,X]**parent[Z,Y]
```

Но это только терминальные случаи. Есть и рекурсивные ветки. Если люди не являются братьями / сестрами, то есть родители у них разные, они находятся в одном поколении, если их родители находятся в одном поколении. Аналогично люди находятся в одном поколении, если их дети находятся в одном поколении:

```
generation[X,Y] |= parent[X,Z]**parent[Y,Z]  |
parent[Z,X]**parent[Z,Y] |
parent[X,Z]**parent[Y,U]**generation[Z,U] |
parent[Z,X]**parent[U,Y]**generation[Z,U]
```

Приведем код программы полностью:

```
from decpy import var, queryclass, table

@queryclass
class person:
    def __init__(self,name,age,mother=None,father=None):
        self.name=name
        self.age=age
        self.mother=mother
        self.father=father
    def __repr__(self):
        return self.name+" "+str(self.age)

X,Y,Z,U=var(4)
mother,father,parent,generation=table(4)
mother[X,Y] |= (person*person)[X,Y,X.mother==Y]
father[X,Y] |= (person*person)[X,Y,X.father==Y]
parent[X,Y] |= mother[X,Y] | father[X,Y]
generation[X,Y] |= parent[X,Z]**parent[Y,Z]  |
parent[Z,X]**parent[Z,Y] |
parent[X,Z]**parent[Y,U]**generation[Z,U] |
parent[Z,X]**parent[U,Y]**generation[Z,U]

p0=person("Ману",1931)
p1=person("Адам",1950,None,p0)
p2=person("Ева",1951)
p3=person("Татьяна",1952)
p4=person("Глеб",1970,p2,p1)
p5=person("Ольга",1970,p3)
p6=person("Иван",1990,p5,p4)
p7=person("Мария",1991)
p8=person("Василий",2000,p7,p6)
p9=person("Светлана",2012,p7,p6)
p10=person("Елена",2012,p5,p4)
p11=person("Роман",2024,p10)
p12=person("Кирилл",2012)
p13=person("Виталий",2024,p7,p12)
p14=person("Ксения",2024)

print("Поколение Елены:")
print(generation[p10,X][None,X])
print("Поколение Романа:")
print(generation[p11,X][None,X])
```

Результат:

```
Поколение Елены:
{Иван 1990, Мария 1991, Елена 2012, Кирилл 2012}
Поколение Романа:
{Роман 2024, Василий 2000, Виталий 2024, Светлана 2012}
```

Этот запрос также зациклился в имеющихся у меня компиляторах Prolog. А библиотека DecPy справилась с рекурсией!

Заметим, что библиотека DecPy работает не вполне так же, как Prolog. Программисты на Prolog любят разбивать определение на куски для каждой ветки:

```
...
generation[X,Y]|=parent[X,Z]**parent[Y,Z]
generation[X,Y]|=parent[Z,X]**parent[Z,Y]
generation[X,Y]|=parent[X,Z]**parent[Y,U]**generation[Z,U]
generation[X,Y]|=parent[Z,X]**parent[U,Y]**generation[Z,U]
...
```

Результат:

```
Поколение Елены:
{Иван 1990, Елена 2012, Мария 1991}
Поколение Романа:
{Роман 2024, Светлана 2012, Василий 2000}
```

Видно, что в поколениях не хватает людей. Это связано с тем, что внутри каждой ветки DecPy ищет все, что может найти, затем переходит к новой ветке и к предыдущей ветке уже не возвращается. Данной задаче это явно вредит, но, возможно, есть задачи, в которых такой эффект полезен. Пока модель рекурсии библиотеки DecPy устроена таким образом. Возможно, в будущем я изменю эту модель.

Выводы

Посмотрим еще раз на определения свояков и поколения и на понятия, на которых они основываются:

```
X,Y,Z,U=var(4)
mother,father,parent,ancestor,rod,svoy,generation=table(7)

mother[X,Y] |= (person*person)[X,Y,X.mother==Y]
father[X,Y] |= (person*person)[X,Y,X.father==Y]
parent[X,Y] |= mother[X,Y] | father[X,Y]
ancestor[X,Y] |= parent[X,Y] | parent[X,Z]**ancestor[Z,Y]
rod[X,Y] |= ancestor[X,Y] | ancestor[Y,X] |
ancestor[X,Z]**ancestor[Y,Z,X!=Y]
svoy[X,Y] |= rod[X,Y] | rod[X,Z]**svoy[Z,Y,X!=Y]

generation[X,Y] |= parent[X,Z]**parent[Y,Z]
| parent[Z,X]**parent[Z,Y]
| parent[X,Z]**parent[Y,U]**generation[Z,U]
| parent[Z,X]**parent[U,Y]**generation[Z,U]
```

Мы видим красивый лаконичный код, в котором нет ничего лишнего! Вряд ли найдется язык, который может похвастать столь элегантными конструкциями! Причем они встроились в язык Python «как родные». Именно ради этого я написал библиотеку DecPy. Но этим ее польза не исчерпывается. Есть еще ряд интересных применений, о которых я хочу рассказать.

5. Запросы к коллекциям переменной длины, пути в графе

Распространенными задачами являются задачи с графами, например, поиска пути. Родословное древо из прошлой главы можно считать разновидностью графа, а задачи, с ним связанные, задачами с графами. Например, поиск свояков – это формирование области достижимости. Но есть важное отличие. Когда мы писали запрос про поиск предка человека, мы рекурсивно получали список предков, но не запоминали линию родства, которая привела нас к нему. Но путь в графе может быть важен. Например, в случае обработки базы полетов самолетов, нам важно не только место, до которого можно добраться (пусть даже с несколькими пересадками), но и конкретные рейсы, на которых мы можем полететь.

5.1. Самолетные рейсы – постановка задачи

Задача. Задана база самолетных рейсов, хранящая записи самолетных рейсов в виде городов отправления и назначения и времени вылетов и прилетов. Надо научиться для двух городов формировать список вариантов перелета из одного города в другой без пересадок, с одной или несколькими пересадками.

Создадим класс для самолетных рейсов и записи о полетах. Для простоты будем считать, что полеты происходят в течение одних суток, не переходя в следующий день, а время будем писать по Москве.

```
from decpy import var, queryclass

@queryclass
class flight:
    def __init__(self,city1,time1,city2,time2):
        self.city1=city1
        self.time1=time1
        self.city2=city2
        self.time2=time2
    def __str__(self):
        return self.city1+" "+str(self.time1)+"
"+self.city2+" "+str(self.time2)
    def __repr__(self):
        return self.city1+" "+str(self.time1)+"
"+self.city2+" "+str(self.time2)

flight("Екатеринбург",0,"Москва",2)
flight("Екатеринбург",1,"Москва",3)
flight("Москва",4,"Екатеринбург",6)
flight("Екатеринбург",1,"Новосибирск",3)
```

```
flight("Москва",3,"Владивосток",10)
flight("Москва",1,"Владивосток",9)
flight("Новосибирск",2,"Владивосток",6)
flight("Новосибирск",4,"Владивосток",8)
flight("Новосибирск",4,"Благовещенск",9)
flight("Благовещенск",10,"Владивосток",13)
flight("Владивосток",14,"Москва",24)

for el in flight:
    print(el)
```

Результат:

```
Екатеринбург 1 Новосибирск 3
Екатеринбург 1 Москва 3
Екатеринбург 0 Москва 2
Новосибирск 4 Благовещенск 9
Москва 3 Владивосток 10
Новосибирск 4 Владивосток 8
Москва 4 Екатеринбург 6
Владивосток 14 Москва 24
Новосибирск 2 Владивосток 6
Благовещенск 10 Владивосток 13
Москва 1 Владивосток 9
```

Создавая маршруты с пересадкой, можно пойти двумя путями, пользуясь разными операторами для декартова произведения:

1) Оператор «*»:

```
flight * flight
```

Пример соединения:

```
(Екатеринбург 1 Новосибирск 3), (Новосибирск 4 Владивосток
8)
```

Такое соединение сохраняет внутреннюю структуру рейсов, хотя и избыточно (в примере Новосибирск повторяется два раза).

2) Оператор «**»:

```
(Екатеринбург 1 Новосибирск 3 Новосибирск 4 Владивосток 8)
```

В таком случае можно будет убрать дубликат при помощи исчисления на доменах:

```
(Екатеринбург 1 Новосибирск 3 4 Владивосток 8)
```

Но представим, что рейсов больше двух, и у них могут быть дополнительные важные атрибуты, например, цена. Думаю, что в маршруте целесообразно сохранить рейсы по отдельности. Схематично это можно представить так:

```
((рейс 1),(рейс 2),(рейс 3),…,(рейс n))
```

До сих пор мы объединяли данные одного формата. В этой задаче возникает проблема соединения полетов без пересадки и полетов с пересадкой в одну коллекцию. Казалось бы, нет проблем:

```
R = A | B
```

Где A – это полеты без пересадки, а B – с пересадкой. Но посмотрим пример такого соединения:

```
{(Москва 3 Владивосток 10),((Екатеринбург 1 Новосибирск 3),
(Новосибирск 4 Владивосток 8))}
```

Если обрабатывать такую структуру с помощью исчисления на кортежах, то при обращении

```
el[0]
```

к первой записи

```
(Москва 3 Владивосток 10)
```

мы получим

```
Москва
```

А при обращении

```
el[0]
```

ко второй записи

```
(Екатеринбург 1 Новосибирск 3), (Новосибирск 4 Владивосток
8)
```

Мы получим

```
(Екатеринбург 1 Новосибирск 3)
```

То есть при одинаковом обращении мы получаем разный по смыслу результат. В случае маршрута из одного рейса мы получаем город отправления, а в случае маршрута из двух рейсов – первый рейс целиком.

Но при сохранении результатов поиска в одной коллекции мы должны получать однотипный результат!

Выход заключается в том, что даже если маршрут состоит из одного рейса, рейс должен быть упакован – он должен быть нулевым (пусть и единственным) элементом записи маршрута. В библиотеке DecPy есть функция упаковки `packelements`. Она, кстати, работает в ленивом режиме:

```
from decpy import var, queryclass, packelements

@queryclass
class flight:
    def __init__(self,city1,time1,city2,time2):
        self.city1=city1
        self.time1=time1
        self.city2=city2
        self.time2=time2
    def __str__(self):
        return self.city1+" "+str(self.time1)+"
"+self.city2+" "+str(self.time2)
    def __repr__(self):
        return self.city1+" "+str(self.time1)+"
"+self.city2+" "+str(self.time2)

R = packelements(flight)

flight("Екатеринбург",0,"Москва",2)
...
flight("Владивосток",14,"Москва",24)

for el in R:
    print(el)
```

Результат:

```
(Новосибирск 4 Благовещенск 9,)
(Новосибирск 2 Владивосток 6,)
(Новосибирск 4 Владивосток 8,)
(Благовещенск 10 Владивосток 13,)
(Екатеринбург 1 Москва 3,)
(Екатеринбург 0 Москва 2,)
(Владивосток 14 Москва 24,)
(Екатеринбург 1 Новосибирск 3,)
(Москва 1 Владивосток 9,)
(Москва 3 Владивосток 10,)
(Москва 4 Екатеринбург 6,)
```

Обратите внимание, как изменилась форма представления результата.

Теперь можно решать задачи, связанные с полетами.

5.2. Рейсы с пересадкой

Задача. Найти рейсы с одной пересадкой.

Это простая задача на декартово произведение с условием соединения. В нашем случае пункт назначения первого рейса должен совпасть с пунктом

вылета второго рейса, причем время окончания первого рейса должно быть меньше времени начала второго рейса:

```
from decpy import var, queryclass, packelements

@queryclass
class flight:
    def __init__(self,city1,time1,city2,time2):
        self.city1=city1
        self.time1=time1
        self.city2=city2
        self.time2=time2
    def __str__(self):
        return self.city1+" "+str(self.time1)+"
"+self.city2+" "+str(self.time2)
    def __repr__(self):
        return self.city1+" "+str(self.time1)+"
"+self.city2+" "+str(self.time2)

f = var()
F = packelements(flight)
R = (F**F)[(f[0].city2==f[1].city1) &
(f[0].time2<f[1].time1)]

flight("Екатеринбург",0,"Москва",2)
...
flight("Владивосток",14,"Москва",24)

for el in R:
    print(el)
```

Результат:

```
(Екатеринбург 0 Москва 2, Москва 4 Екатеринбург 6)
(Москва 3 Владивосток 10, Владивосток 14 Москва 24)
(Новосибирск 4 Благовещенск 9, Благовещенск 10 Владивосток
13)
(Екатеринбург 0 Москва 2, Москва 3 Владивосток 10)
(Екатеринбург 1 Новосибирск 3, Новосибирск 4 Благовещенск
9)
(Екатеринбург 1 Москва 3, Москва 4 Екатеринбург 6)
(Екатеринбург 1 Новосибирск 3, Новосибирск 4 Владивосток 8)
(Новосибирск 2 Владивосток 6, Владивосток 14 Москва 24)
(Москва 1 Владивосток 9, Владивосток 14 Москва 24)
(Благовещенск 10 Владивосток 13, Владивосток 14 Москва 24)
(Новосибирск 4 Владивосток 8, Владивосток 14 Москва 24)
```

Если внимательно посмотреть на рейсы, то можно увидеть круговые полеты: летя с пересадкой, мы возвращаемся в первый город, из которого вылетели.

Сделаем дополнительную проверку, чтобы это исключить (это поможет нам в будущем исключить петли при перелетах с неограниченным количеством пересадок):

```
...
R = (F**F)[(f[0].city2==f[1].city1) &
(f[0].time2<f[1].time1) & (f[0].city1!=f[1].city2)]
...
```

Результат:

```
(Благовещенск 10 Владивосток 13, Владивосток 14 Москва 24)
(Новосибирск 4 Владивосток 8, Владивосток 14 Москва 24)
(Екатеринбург 1 Новосибирск 3, Новосибирск 4 Владивосток 8)
(Екатеринбург 0 Москва 2, Москва 3 Владивосток 10)
(Екатеринбург 1 Новосибирск 3, Новосибирск 4 Благовещенск
9)
(Новосибирск 2 Владивосток 6, Владивосток 14 Москва 24)
(Новосибирск 4 Благовещенск 9, Благовещенск 10 Владивосток
13)
```

Количество рейсов уменьшилось. Соединим теперь рейсы без пересадки с рейсами с пересадкой:

```
...
R = F | (F**F)[(f[0].city2==f[1].city1) &
(f[0].time2<f[1].time1) & (f[0].city1!=f[1].city2)]
...
```

Результат:

```
(Екатеринбург 0 Москва 2,)
(Новосибирск 4 Благовещенск 9,)
(Владивосток 14 Москва 24,)
(Москва 4 Екатеринбург 6,)
(Екатеринбург 1 Новосибирск 3,)
(Новосибирск 2 Владивосток 6, Владивосток 14 Москва 24)
(Москва 3 Владивосток 10,)
(Екатеринбург 0 Москва 2, Москва 3 Владивосток 10)
(Екатеринбург 1 Новосибирск 3, Новосибирск 4 Благовещенск
9)
(Благовещенск 10 Владивосток 13,)
(Благовещенск 10 Владивосток 13, Владивосток 14 Москва 24)
(Новосибирск 4 Владивосток 8, Владивосток 14 Москва 24)
(Екатеринбург 1 Москва 3,)
(Новосибирск 2 Владивосток 6,)
(Москва 1 Владивосток 9,)
(Новосибирск 4 Владивосток 8,)
(Екатеринбург 1 Новосибирск 3, Новосибирск 4 Владивосток 8)
(Новосибирск 4 Благовещенск 9, Благовещенск 10 Владивосток
13)
```

5.3. Рейсы с неограниченным количеством пересадок

Очевидно, чтобы сформировать множество рейсов с любым количеством пересадок, надо запрос:

```
R = F | (F**F)[(f[0].city2==f[1].city1) &
(f[0].time2<f[1].time1) & (f[0].city1!=f[1].city2)]
```

сделать рекурсивным, наподобие того, как мы от определения понятия «дедушка» перешли к понятию «предок». Вспомним этот переход:

```
grandparent[X,Y] |= parent[X,Z]**parent[Z,Y]
ancestor[X,Y] |= parent[X,Y] | parent[X,Z]**ancestor[Z,Y]
```

Заменим оператор «|» на «|=», предварительно объявим `R` как `var` (`table` здесь не годится), заменим второй (правый) множитель декартова произведения `F` на `R`:

```
R = var()
R |= F | (F**R)[(f[0].city2==f[1].city1) &
(f[0].time2<f[1].time1) & (f[0].city1!=f[1].city2)]
```

Осталось сделать еще одну замену: пункт отправления первого (нулевого) рейса не должен совпасть с пунктом прибытия последнего рейса. И здесь исчисление на кортежах очень уместно. Мы можем обращаться к каждому перелету записи по её номеру, используя, в том числе, обратную индексацию:

```
R = var()
R |= F | (F**R)[(f[0].city2==f[1].city1) &
(f[0].time2<f[1].time1) & (f[0].city1!=f[-1].city2)]
```

Проверим работу, выведя все возможные варианты перелетов из Екатеринбурга во Владивосток:

```
from decpy import var, queryclass, packelements

@queryclass
class flight:
    def __init__(self,city1,time1,city2,time2):
        self.city1=city1
        self.time1=time1
        self.city2=city2
        self.time2=time2
    def __str__(self):
        return self.city1+" "+str(self.time1)+"
"+self.city2+" "+str(self.time2)
    def __repr__(self):
        return self.city1+" "+str(self.time1)+"
"+self.city2+" "+str(self.time2)
```

```
f = var()
F = packelements(flight)
R = var()
R |= F | (F**R)[(f[0].city2==f[1].city1) &
(f[0].time2<f[1].time1) & (f[0].city1!=f[-1].city2)]

flight("Екатеринбург",0,"Москва",2)
flight("Екатеринбург",1,"Москва",3)
flight("Москва",4,"Екатеринбург",6)
flight("Екатеринбург",1,"Новосибирск",3)
flight("Москва",3,"Владивосток",10)
flight("Москва",1,"Владивосток",9)
flight("Новосибирск",2,"Владивосток",6)
flight("Новосибирск",4,"Владивосток",8)
flight("Новосибирск",4,"Благовещенск",9)
flight("Благовещенск",10,"Владивосток",13)
flight("Владивосток",14,"Москва",24)

for el in R[(f[0].city1=="Екатеринбург") & (f[-
1].city2=="Владивосток")]:
    print(el)
```

Результат:

```
(Екатеринбург 1 Новосибирск 3, Новосибирск 4 Благовещенск
9, Благовещенск 10 Владивосток 13)
(Екатеринбург 1 Новосибирск 3, Новосибирск 4 Владивосток 8)
(Екатеринбург 0 Москва 2, Москва 3 Владивосток 10)
```

На основе созданной коллекции рейсов с неограниченным количеством пересадок можно решать разные задачи, например, вывести рейсы с двумя пересадками:

```
...
M = R[f.len()==3]
for el in M:
    print(M)
```

Результат:

```
{(Новосибирск 4 Благовещенск 9, Благовещенск 10 Владивосток
13, Владивосток 14 Москва 24), (Екатеринбург 1 Новосибирск
3, Новосибирск 4 Благовещенск 9, Благовещенск 10
Владивосток 13), (Екатеринбург 1 Новосибирск 3, Новосибирск
4 Владивосток 8, Владивосток 14 Москва 24)}
{(Новосибирск 4 Благовещенск 9, Благовещенск 10 Владивосток
13, Владивосток 14 Москва 24), (Екатеринбург 1 Новосибирск
3, Новосибирск 4 Благовещенск 9, Благовещенск 10
Владивосток 13), (Екатеринбург 1 Новосибирск 3, Новосибирск
4 Владивосток 8, Владивосток 14 Москва 24)}
```

```
{(Новосибирск 4 Благовещенск 9, Благовещенск 10 Владивосток
13, Владивосток 14 Москва 24), (Екатеринбург 1 Новосибирск
3, Новосибирск 4 Благовещенск 9, Благовещенск 10
Владивосток 13), (Екатеринбург 1 Новосибирск 3, Новосибирск
4 Владивосток 8, Владивосток 14 Москва 24)}
```

Узнаем наибольшее число пересадок:

```
...
print(R[f.len()].max())
```

Результат:

```
4
```

Узнаем, что это за перелет:

```
...
print(R[f].max(f.len()))
```

Результат:

```
(Екатеринбург 1 Новосибирск 3, Новосибирск 4 Благовещенск
9, Благовещенск 10 Владивосток 13, Владивосток 14 Москва
24)
```

5.4. Суммарная стоимость перелетов

Добавим цену к каждому перелету и посчитаем суммарную стоимость каждого варианта перелета из Екатеринбурга во Владивосток. Приведем код программы полностью:

```
from decpy import var, queryclass, packelements

@queryclass
class flight:
    def __init__(self,city1,time1,city2,time2,price):
        self.city1=city1
        self.time1=time1
        self.city2=city2
        self.time2=time2
        self.price=price
    def __str__(self):
        return self.city1+" "+str(self.time1)+"
"+self.city2+" "+str(self.time2)
    def __repr__(self):
        return self.city1+" "+str(self.time1)+"
"+self.city2+" "+str(self.time2)+" "+str(self.price)

f = var()
F = packelements(flight)
R = var()
```

```
R |= F | (F**R)[(f[0].city2==f[1].city1) &
(f[0].time2<f[1].time1) & (f[0].city1!=f[-1].city2)]

flight("Екатеринбург",0,"Москва",2,10000)
flight("Екатеринбург",1,"Москва",3,15000)
flight("Москва",4,"Екатеринбург",6,20000)
flight("Екатеринбург",1,"Новосибирск",3,25000)
flight("Москва",3,"Владивосток",10,90000)
flight("Москва",1,"Владивосток",9,95000)
flight("Новосибирск",2,"Владивосток",6,40000)
flight("Новосибирск",4,"Владивосток",8,45000)
flight("Новосибирск",4,"Благовещенск",9,50000)
flight("Благовещенск",10,"Владивосток",13,30000)
flight("Владивосток",14,"Москва",24,100000)

for el in R[(f[0].city1=="Екатеринбург") & (f[-
1].city2=="Владивосток")]:
    print(el, var(el)[f.price].sum())
```

Результат:

```
(Екатеринбург 1 Новосибирск 3 25000, Новосибирск 4
Владивосток 8 45000) 70000
(Екатеринбург 0 Москва 2 10000, Москва 3 Владивосток 10
90000) 100000
(Екатеринбург 1 Новосибирск 3 25000, Новосибирск 4
Благовещенск 9 50000, Благовещенск 10 Владивосток 13 30000)
105000
```

Заметьте, что здесь в цикле мы применили запрос уже к выбранному элементу коллекции:

```
var(el)[f.price].sum()
```

Теперь гипотетический пользователь может выбрать рейс, оценив пересадки и суммарную стоимость. Но что, если мы хотим, чтобы система сама вывела вариант с наименьшей суммарной стоимостью независимо от количества пересадок?

Очевидно, нам нужно обработать таблицу `R`, добавив к ней колонку суммарной стоимости рейсов. Несмотря на очевидность, это будет плохое (сложное) решение. Тем не менее, приведем сперва его, а потом более простое (изящное) решение.

Покажем, как это сделать, на более простом абстрактном примере.

Добавление дополнительного поля к простой коллекции делается просто:

```
from decpy import var
el = var()
L = var([1,2,3,4,5])
M = L[el,el**2]
print(M)
```

Результат:

```
{(2, 4), (4, 16), (1, 1), (3, 9), (5, 25)}
```

Но если мы применим это для составной коллекции:

```
from decpy import var, packelements, table

el = var()
L,M,P=table(3)
L |= var({(1,2,3),(4,5,6),(7,8,9)})
M |= L[el,el.sum()]
for el in P:
    print(el)
```

то мы неожиданно получим пустой ответ. Дело в том, что библиотека DecPy восприняла это как запрос в стиле исчисления на доменах и пытается для первого кортежа `el` применить к нулевым элементам, а `el.sum` к первым элементам. Естественно, это не получается.

Поправить положение можно, если глубже упаковать весь кортеж, чтобы за `el` библиотека понимала его целиком – `(1,2,3)`. Сделать это можно с помощью уже встречавшейся нам функции `packelements`:

```
from decpy import var, packelements, table
el = var()
L,M,P=table(3)
L |= var({(1,2,3),(4,5,6),(7,8,9)})
M |= packelements(L)
for el in M:
    print(el)
el = var()
P |= M[el,el[0].sum()]
for el in P:
    print(el)
```

Результат:

```
((7, 8, 9),)
((1, 2, 3),)
((4, 5, 6),)
(((7, 8, 9),), 24)
(((4, 5, 6),), 15)
(((1, 2, 3),), 6)
```

Уберем теперь излишнее вложение:

```
from decpy import var, packelements, table

el = var()
L,M,P=table(3)
L |= var({(1,2,3),(4,5,6),(7,8,9)})
M |= packelements(L)
el,a,b = var(3)
P |= M[el,el[0].sum()][a[0],b]
for el in P:
    print(el)
```

Вернемся к задаче с самолетными рейсами и добавим к каждому маршруту суммарную стоимость входящих в него полетов.

```
from decpy import var, queryclass, packelements, table

@queryclass
class flight:
    def __init__(self,city1,time1,city2,time2,price):
        self.city1=city1
        self.time1=time1
        self.city2=city2
        self.time2=time2
        self.price=price
    def __str__(self):
        return self.city1+" "+str(self.time1)+"
"+self.city2+" "+str(self.time2)
    def __repr__(self):
        return self.city1+" "+str(self.time1)+"
"+self.city2+" "+str(self.time2)+" "+str(self.price)

f = var()
F = packelements(flight)
R = var()
R |= F | (F**R)[(f[0].city2==f[1].city1) &
(f[0].time2<f[1].time1) & (f[0].city1!=f[-1].city2)]

flight("Екатеринбург",0,"Москва",2,10000)
...
flight("Владивосток",14,"Москва",24,100000)

el,a,b = var(3)
S = packelements(R)
P = S[el,el[0][f.price].sum()][a[0],b]
T = P[a,b,(a[0].city1=="Екатеринбург") & (a[-
1].city2=="Владивосток")]
for el in T:
    print(el)
```

Результат:

```
((Екатеринбург 0 Москва 2 10000, Москва 3 Владивосток 10
90000), 100000)
((Екатеринбург 1 Новосибирск 3 25000, Новосибирск 4
Владивосток 8 45000), 70000)
((Екатеринбург 1 Новосибирск 3 25000, Новосибирск 4
Благовещенск 9 50000, Благовещенск 10 Владивосток 13
30000), 105000)
```

Выведем теперь рейс с минимальной ценой:

```
from decpy import var, queryclass, packelements, table

@queryclass
class flight:
    def __init__(self,city1,time1,city2,time2,price):
        self.city1=city1
        self.time1=time1
        self.city2=city2
        self.time2=time2
        self.price=price
    def __str__(self):
        return self.city1+" "+str(self.time1)+"
"+self.city2+" "+str(self.time2)
    def __repr__(self):
        return self.city1+" "+str(self.time1)+"
"+self.city2+" "+str(self.time2)+" "+str(self.price)

f = var()
F = packelements(flight)
R = var()
R |= F | (F**R)[(f[0].city2==f[1].city1) &
(f[0].time2<f[1].time1) & (f[0].city1!=f[-1].city2)]

flight("Екатеринбург",0,"Москва",2,10000)
...
flight("Владивосток",14,"Москва",24,100000)

el,a,b = var(3)
S = packelements(R)
P = S[el,el[0][f.price].sum()][a[0],b]
T = P[a,b,(a[0].city1=="Екатеринбург") & (a[-
1].city2=="Владивосток")].min(el[1])
print(T)
```

Результат:

```
((Екатеринбург 1 Новосибирск 3 25000, Новосибирск 4
Владивосток 8 45000), 70000)
```

Но можно найти самый дешевый вариант намного проще. Вводить дополнительный атрибут – это очень громоздкий вариант, который стоит делать в самом крайнем случае. Вспомним, что метод `min` достаточно интеллектуален, туда можно написать критерий отбора. И этот критерий сам по себе будет являться запросом! Приведем код полностью:

```
from decpy import var, queryclass, packelements, table

@queryclass
class flight:
    def __init__(self,city1,time1,city2,time2,price):
        self.city1=city1
        self.time1=time1
        self.city2=city2
        self.time2=time2
        self.price=price
    def __str__(self):
        return self.city1+" "+str(self.time1)+"
"+self.city2+" "+str(self.time2)
    def __repr__(self):
        return self.city1+" "+str(self.time1)+"
"+self.city2+" "+str(self.time2)+" "+str(self.price)

f = var()
F = packelements(flight)
R = var()
R |= F | (F**R)[(f[0].city2==f[1].city1) &
(f[0].time2<f[1].time1) & (f[0].city1!=f[-1].city2)]

flight("Екатеринбург",0,"Москва",2,10000)
flight("Екатеринбург",1,"Москва",3,15000)
flight("Москва",4,"Екатеринбург",6,20000)
flight("Екатеринбург",1,"Новосибирск",3,25000)
flight("Москва",3,"Владивосток",10,90000)
flight("Москва",1,"Владивосток",9,95000)
flight("Новосибирск",2,"Владивосток",6,40000)
flight("Новосибирск",4,"Владивосток",8,45000)
flight("Новосибирск",4,"Благовещенск",9,50000)
flight("Благовещенск",10,"Владивосток",13,30000)
flight("Владивосток",14,"Москва",24,100000)

el=var()
P = R[(el[0].city1=="Екатеринбург") &
(el[-1].city2=="Владивосток")].min(el[f.price].sum())
print(P)
```

Результат:

```
(Екатеринбург 1 Новосибирск 3 25000, Новосибирск 4
Владивосток 8 45000)
```

Еще раз приведем основной код программы, состоящий из двух запросов.

1) Формирование всех маршрутов с любым количеством пересадок:

```
R |= F | (F**R)[(f[0].city2==f[1].city1) &
(f[0].time2<f[1].time1) & (f[0].city1!=f[-1].city2)]
```

2) Выборка самого дешевого варианта между двумя городами :

```
P = R[(el[0].city1=="Екатеринбург") &
(el[-1].city2=="Владивосток")].min(el[f.price].sum())
```

Код предельно лаконичен!

Выводы

Пример задачи на полеты самолетов отлично демонстрирует сочетание разных стилей декларативного программирования. Здесь используется исчисление на кортежах (причем кортежи имеют переменную длину), рекурсивный запрос, прямая и обратная адресация Python (прямую и обратную адресацию Python, а также срезы можно считать прообразом встроенного в Python декларативного программирования). Это доказывает пользу сочетания разных видов декларативного программирования в рамках единых языковых конструкций.

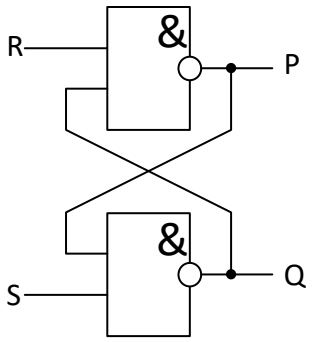
6. Интеллектуальные задачи

В предыдущих четырех разделах мы показали основное применение библиотеки DecPy – запросы, как в базах данных, по аналогии с SQL и QBE, формулирование понятий, как в языке Prolog, рекурсивные запросы к графовым структурам данных. В этой главе покажем, что DecPy справляется с задачами Prolog, которые характеризуют его как язык искусственного интеллекта.

6.1. Логические элементы

Название языка Prolog расшифровывается как «programming in logic» - программирование в логике. А какая же логика без логических элементов из схемотехники?

Смоделируем простейшее устройство памяти – ***RS-триггер***, состоящий из двух элементов `notand`:



Сделаем логический элемент `And` («и»), который, фактически, будет представлять собой сохраненную таблицу истинности:

```
from decpy import var,table
And=table()
And(0,0,0)
And(0,1,0)
And(1,0,0)
And(1,1,1)
print(And)
```

Результат:

```
{(1, 0, 0), (0, 0, 0), (1, 1, 1), (0, 1, 0)}
```

Здесь первые два числа в кортеже ответа – это входные данные (сигналы), а последнее число – ответ.

К элементу `And` мы можем задавать вопросы в стиле Prolog. Например, что получится, если на вход подать сигналы 0, 1 ?

```
from decpy import var,table
And=table()
And(0,0,0)
And(0,1,0)
And(1,0,0)
And(1,1,1)
z=var()
print(And[0,1,z][None,None,z])
```

Результат:

```
{0}
```

Какими должны быть входные сигналы, чтобы получить на выходе 0?

```
from decpy import var,table
And=table()
And(0,0,0)
And(0,1,0)
And(1,0,0)
And(1,1,1)
x,y=var(2)
print(And[x,y,0][x,y,None])
```

Результат:

```
{(1, 0), (0, 0), (0, 1)}
```

Каким должен быть второй входной сигнал, чтобы при первом входном сигнале, равном 1, на выходе получился 0?

```
from decpy import var,table
And=table()
And(0,0,0)
And(0,1,0)
And(1,0,0)
And(1,1,1)
x,y=var(2)
print(And[1,y,0][None,y,None])
```

Результат:

```
{0}
```

Аналогично `And` сделаем элемент `Not` («не»):

```
from decpy import var,table
Not=table()
Not(0,1)
Not(1,0)
print(Not)
```

Результат:

```
{(0, 1), (1, 0)}
```

На основе And и Not сделаем элемент Notand:

```
from decpy import var,table
And,Not,Notand=table(3)
And(0,0,0)
And(0,1,0)
And(1,0,0)
And(1,1,1)
Not(0,1)
Not(1,0)
x,y,z,v=var(4)
Notand[x,y,v]|=And[x,y,z]**Not[z,v]
print(Notand)
```

Результат:

```
{(1, 0, 1), (0, 0, 1), (1, 1, 0), (0, 1, 1)}
```

Наконец, создадим RS-триггер, увязывая входные и выходные переменные двух элементов Notand:

```
from decpy import var,table
And,Not,Notand,Trigger=table(4)
And(0,0,0)
And(0,1,0)
And(1,0,0)
And(1,1,1)
Not(0,1)
Not(1,0)
x,y,z,v=var(4)
Notand[x,y,v]|=And[x,y,z]**Not[z,v]
r,s,p,q = var(4)
Trigger[r,s,p,q] |= Notand[r,q,p]**Notand[p,s,q]
print(Trigger)
```

Результат:

```
{(1, 1, 1, 0), (0, 1, 1, 0), (1, 0, 0, 1), (1, 1, 0, 1),
(0, 0, 1, 1)}
```

Несмотря на очевидную простоту моделирования логических элементов, Prolog не получил распространения в схемотехнике по причине медленной работы.

6.2. Таблица умножения

Задача. Дано число. Получить все варианты разложения заданного числа на два множителя.

Подобно тому, как мы поступили с логическим элементом `And` в прошлой главе, составим таблицу умножения: первыми двумя элементами таблицы будут множители, последним – результат. Например, `2*3=6` будет записано как `(2,3,6)`.

Для составления таблицы нам понадобится ряд чисел до заданного числа. Определить его можно несколькими способами:

```
from decpy import var, table
n=5
number=table()
for i in range(1,n+1):
    number(i)
print(number)
```

Результат:

```
{1, 2, 3, 4, 5}
```

Второй способ заключается в преобразовании `range` во множество и сохранении его в ленивой переменной:

```
from decpy import var, table
n=5
number=var(set(range(1,n+1)))
print(number)
```

Результат:

```
{1, 2, 3, 4, 5}
```

Генерация различных числовых последовательностей является распространенной задачей, и в библиотеке DecPy есть функция `lazyrange` с богатым набором возможностей. Работает она похоже на `range`, но является не генератором, который постепенно выдает ответы, а ленивой функцией. Посмотрим на примеры ее использования:

```
from decpy import var,lazyrange

print(lazyrange(10))
print(lazyrange(4,10))
print(lazyrange(4,10,2))
print(lazyrange(10,4,-2))
print(lazyrange(10,4))
print(lazyrange(4,10,0.5))
print(lazyrange(1,100,lambda n:n*2))
n=var()
print(lazyrange(1,100,n*2))
a,b=var(2)
print(lazyrange(0,1,100,lambda a,b:a+b))
```

Результат:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[4, 5, 6, 7, 8, 9]
[4, 6, 8]
[10, 8, 6]
[10, 9, 8, 7, 6, 5]
[4, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5]
[1, 2, 4, 8, 16, 32, 64]
[1, 2, 4, 8, 16, 32, 64]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Проанализируем результат. Первые три примера работают точно так же, как и `range` в Python

```
print(lazyrange(10))
print(lazyrange(4,10))
print(lazyrange(4,10,2))
print(lazyrange(10,4,-2))
```

выдавая ожидаемые результаты:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[4, 5, 6, 7, 8, 9]
[4, 6, 8]
[10, 8, 6]
```

В пятом примере:

```
print(lazyrange(10,4))
```

в случае, если бы это был `range` из Python, программа ушла бы в бесконечный цикл. `lazyrange` же автоматически определяет, что надо не увеличивать, а уменьшать числа:

```
[10, 9, 8, 7, 6, 5]
```

что позволяет избежать типичной ошибки новичков по невнимательности.

Посмотрим на шестой пример:

```
print(lazyrange(4,10,0.5))
```

Оказывается, что `lazyrange`, в отличие от `range` из Python, способна работать не только с целыми числами, но и дробными, задавая, например, дробный шаг:

```
[4, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5]
```

`range` в Python - это, по сути, арифметическая прогрессия, в которой каждый следующий элемент на заданное число (шаг) больше или меньше предыдущего. `lazyrange` позволяет формировать другие прогрессии,

задавая вместо третьего аргумента (шага) функцию изменения (такая же возможность есть в языках семейства C++). Примеры 7 и 8 показывают генерацию геометрической прогрессии, в котором каждый следующий элемент больше предыдущего во сколько-то раз (в примере – в 2):

```
[1, 2, 4, 8, 16, 32, 64]
```

В 7 примере прогрессия задается с помощью анонимной функции `lambda` (функциональный стиль):

```
print(lazyrange(1,100,lambda n:n*2))
```

А в 8 – с помощью ленивой переменной:

```
n=var()
print(lazyrange(1,100,n*2))
```

Последний пример:

```
a,b=var(2)
print(lazyrange(0,1,100,lambda a,b:a+b))
```

задает знаменитую последовательность чисел Фибоначчи:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Здесь старт состоит не из одного числа, а из двух: 0 и 1. Каждое последующее число задается суммой двух предыдущих. `lazyrange` не ограничивает количество стартовых переменных и позволяет задавать различные формулы над ними, позволяя генерировать большое количество разных последовательностей.

Вернемся к таблице умножения и сформируем ее:

```
from decpy import var, table, lazyrange

n=61
x,y,z=var(3)
number=var(set(lazyrange(2,n)))
mult=table()
mult|=(number*number*number)[x,y,z,z==x*y]
print(mult)
```

Результат:

```
{(30, 2, 60), (12, 4, 48), (7, 8, 56), (9, 4, 36), (5, 4,
20), (9, 2, 18), (2, 23, 46), (4, 8, 32), (4, 12, 48), (6,
7, 42), (9, 6, 54), (12, 3, 36), (3, 11, 33), (4, 14, 56),
(4, 5, 20), (8, 3, 24), (3, 4, 12), (5, 8, 40), (6, 6, 36),
(11, 3, 33), (2, 8, 16), (11, 2, 22), (4, 11, 44), (26, 2,
52), (3, 17, 51), (18, 2, 36), (2, 15, 30), (6, 5, 30), (3,
10, 30), (4, 7, 28), (3, 12, 36), (23, 2, 46), (2, 22, 44),
(3, 5, 15), (7, 4, 28), (28, 2, 56), (5, 2, 10), (4, 4,
16), (2, 13, 26), (5, 12, 60), (2, 3, 6), (16, 3, 48), (25,
2, 50), (2, 2, 4), (2, 20, 40), (6, 4, 24), (2, 30, 60),
(2, 10, 20), (3, 18, 54), (9, 3, 27), (2, 27, 54), (7, 6,
42), (4, 10, 40), (12, 5, 60), (22, 2, 44), (5, 9, 45),
(14, 2, 28), (20, 3, 60), (9, 5, 45), (27, 2, 54), (7, 7,
49), (19, 2, 38), (2, 25, 50), (5, 6, 30), (2, 7, 14), (15,
4, 60), (24, 2, 48), (5, 5, 25), (4, 3, 12), (16, 2, 32),
(2, 14, 28), (2, 5, 10), (21, 2, 42), (3, 7, 21), (6, 9,
54), (2, 12, 24), (3, 20, 60), (8, 4, 32), (14, 3, 42),
(13, 4, 52), (10, 2, 20), (2, 19, 38), (3, 13, 39), (6, 8,
48), (3, 15, 45), (10, 3, 30), (4, 6, 24), (15, 2, 30), (3,
6, 18), (11, 5, 55), (7, 2, 14), (3, 8, 24), (2, 26, 52),
(11, 4, 44), (20, 2, 40), (4, 2, 8), (2, 17, 34), (6, 3,
18), (12, 2, 24), (6, 10, 60), (10, 4, 40), (17, 2, 34),
(2, 24, 48), (13, 3, 39), (3, 19, 57), (5, 3, 15), (19, 3,
57), (17, 3, 51), (10, 5, 50), (8, 6, 48), (3, 14, 42),
(15, 3, 45), (29, 2, 58), (3, 16, 48), (6, 2, 12), (8, 7,
56), (2, 4, 8), (7, 5, 35), (4, 13, 52), (18, 3, 54), (2,
21, 42), (3, 9, 27), (5, 11, 55), (8, 5, 40), (2, 29, 58),
(2, 11, 22), (3, 2, 6), (5, 10, 50), (14, 4, 56), (4, 9,
36), (2, 28, 56), (10, 6, 60), (8, 2, 16), (2, 18, 36), (4,
15, 60), (5, 7, 35), (7, 3, 21), (2, 9, 18), (13, 2, 26),
(2, 6, 12), (3, 3, 9), (2, 16, 32)}
```

Теперь с помощью таблицы умножения можно решать различные задачи. Например, получим все разложения числа 60 на пары множителей:

```
from decpy import var, table, lazyrange

n=61
x,y,z=var(3)
number=var(set(lazyrange(2,n)))
mult=table()
mult|=(number*number*number)[x,y,z,z==x*y]
print(mult[x,y,60])
```

Результат:

```
{(30, 2, 60), (4, 15, 60), (5, 12, 60), (10, 6, 60), (12,
5, 60), (6, 10, 60), (3, 20, 60), (15, 4, 60), (2, 30, 60),
(20, 3, 60)}
```

Уберем дубликаты, связанные с переменой мест слагаемых, и не будем выводить само число 60:

```
from decpy import var, table, lazyrange

n=61
x,y,z=var(3)
number=var(set(lazyrange(2,n)))
mult=table()
mult|=(number*number*number)[x,y,z,z==x*y]
print(mult[x,y,60,x<=y][x,y,None])
```

Результат:

```
{(3, 20), (4, 15), (6, 10), (5, 12), (2, 30)}
```

Есть альтернатива заданию таблицы умножения:

```
mult|=(number*number*number)[x,y,z,z==x*y]
```

Вспомним, что хорошим стилем является описание атрибутов сразу после имени таблицы (аналог исчисления предикатов первого порядка):

```
mult[x,y,z]
```

Можно определить таблицу умножения в стиле исчисления предикатов первого порядка:

```
mult[x,y,z]|=number[x,]**number[y,]**number[z,z==x*y]
```

Обратите внимание на замену оператора «*» на «**». Кроме того, пришлось ставить запятые в трех местах квадратных скобок: `[x,  ]`. Это не очень красиво, но иначе DecPy воспримет запись как исчисление на кортежах, ведь `number` состоит только из одного элемента.

6.3. Конструирование функций через запросы

Следующий пример искусственный (не стоит так делать в реальном программировании), но через запросы мы можем конструировать функции. Рассмотрим факториал:

```
n! = 1*2*3*…*(n-1)*n
```

Например, `5!=1*2*3*4*5=120`

Есть рекурсивный алгоритм вычисления факториала. Заметим, что `5!=4!*5`.

В общем виде алгоритм таков:

`1!=1` – терминальная ветка;

`n!=(n-1)!*n` – ветка с рекурсией.

Легко написать вычисление факториала через цикл или рекурсивную функцию. Мы же сконструируем факториал через запросы.

Выберем рекурсивную версию факториала. Заметим, что для вычисления факториала нужен инкремент: `n=n+1` через который мы найдем `n-1`, и умножение.

Сделаем их в виде запросов, убедимся, что они работают правильно:

```
from decpy import var, table, lazyrange
n=120
number=var(set(lazyrange(n+1)))
x,y,z=var(3)
inc,mult=table(2)
inc=(number*number)[x,y,y==x+1]
mult|=(number*number*number)[x,y,z,z==x*y]
print(inc[5,y])
print(inc[x,9])
print(mult[x,y,12])
```

Результат:

```
{(5, 6)}
{(8, 9)}
{(4, 3, 12), (6, 2, 12), (3, 4, 12), (12, 1, 12), (2, 6,
12), (1, 12, 12)}
```

Сконструируем теперь факториал. Объявим факториал как таблицу:

```
fact=table()
```

Напишем терминальный случай:

```
fact(1,1)
```

Объявим переменные и начнем писать общий случай:

```
n,r=var(2)
fact(1,1)
fact[n,r]|=
```

Для вычисления нам понадобится `n-1`. Введем переменную `m=n-1` и вычислим ее через инкремент:

```
n,r,m=var(3)
fact(1,1)
fact[n,r]|=inc[m,n]
```

Далее нам понадобится вычислить m! Обозначим m! переменной x. Появилась рекурсия:

```
n,r,m,x=var(4)
fact(1,1)
fact[n,r]|=inc[m,n]**fact[m,x]
```

Заметим, что здесь «**» - это не возведение ответов в степень, а декартово произведение таблиц.

Наконец, надо умножить значение x (факториал от n-1) на значение переменной x, и мы получим ответ, то есть переменную r:

```
fact[n,r]|=inc[m,n]**fact[m,x]** mult[x,n,r]
```

Приведем программу полностью с демонстрацией использования факториала:

```
from decpy import var, table, lazyrange, lazyset
n=25
number=var(set(lazyrange(n+1)))
x,y,z = var(3)
inc,mult,fact = table(3)
inc=(number*number)[x,y,y==x+1]
mult|=(number*number*number)[x,y,z,z==x*y]
n,r,m,x = var(4)
fact(1,1)
fact[n,r]|=inc[m,n]**fact[m,x]**mult[x,n,r]
print(fact())
print(fact()[4,r])
print(fact()[n,24])
```

Результат:

```
{(1, 1), (2, 2), (4, 24), (3, 6)}
{(4, 24)}
{(4, 24)}
```

6.4. Функции как таблицы для запросов

Функции, сконструированные через запросы, работают медленно. Но задачи, связанные с поиском по таблицам, составленным на основе вычислений функций, не такая уж и редкость. Подобно тому, как декоратор queryclass автоматически дополняет класс его экстентом (коллекцией для хранения его экземпляров), в библиотеке DecPy есть декоратор queryfun, сохраняющий аргументы функции вместе с результатом её выполнения.

Сформируем таблицу умножения с помощью queryfun:

```
from decpy import var, lazyrange, queryfun
```

```
@queryfun
def mult(x,y):
    return x*y

mult.init(lazyrange(7),lazyrange(7))
print(mult(3,4))
x,y,z=var(3)
print(mult[3,4,z])
print(mult[3,y,12])
print(mult[x,y,12])
print(mult[3,4,12])
print(mult[3,4,15])
```

Результат:

```
12
{3, 4, 12}
{(3, 4, 12)}
{(4, 3, 12), (2, 6, 12), (3, 4, 12), (6, 2, 12)}
True
False
```

Здесь `init` заполняет таблицу функции, инициализируя аргументы областью определения функции.

Заметим, что `mult` продолжает работать и как обычная функция, и как таблица для запросов.

6.5. Поиск операций по числам

Можно придумать задачи, в которых неизвестными будут не числа, а операции. Например, заданы операции сложения и умножения. Какими способами можно получить 6 из чисел 1,2,3? Есть два способа:

```
1+2+3=6
1*2*3=6
```

Решим эту задачу. Вместо отдельных таблиц для сложения и умножения создадим единую таблицу `operation`, в которой, помимо операндов и результата операции, сохраним тип операции - сложение или умножение:

```
from decpy import var, table, lazyrange
number=var(set(lazyrange(1,5)))
x,y,z=var(3)
operation,formula=table(2)
operation|=(number*number*number)[x,y,z,z==x*y][x,y,z,"*"]
operation|=(number*number*number)[x,y,z,z==x+y][x,y,z,"+"]
print(operation)
```

Результат:

```
{(2, 2, 4, '+'), (1, 1, 2, '+'), (1, 4, 4, '*'), (1, 3, 3,
'*'), (1, 1, 1, '*'), (2, 2, 4, '*'), (1, 2, 2, '*'), (1,
3, 4, '+'), (3, 1, 4, '+'), (1, 2, 3, '+'), (3, 1, 3, '*'),
(2, 1, 3, '+'), (4, 1, 4, '*'), (2, 1, 2, '*')}
```

Теперь нужно сконструировать формулу:

```
x @ y @ z = r
```

где вместо `@` будут подставляться операции «+» или «*»:

```
formula=table()
```

Нам понадобятся переменные для чисел `x`, `y`, `z`, `r` и переменная `s` для промежуточного результата `x @ y`. Еще две переменных, `op1` и `op2`, будут означать подставляемые операции. Сама формула будет декартовым произведением двух операций:

```
from decpy import var, table, lazyrange
number=var(set(lazyrange(1,9)))
x,y,z=var(3)
operation,formula=table(2)
operation|=(number*number*number)[x,y,z,z==x*y][x,y,z,"*"]
operation|=(number*number*number)[x,y,z,z==x+y][x,y,z,"+"]
r,s,op1,op2=var(4)
formula[x,y,z,r,op1,op2]|=operation[x,y,s,op1]**operation[s
,z,r,op2]
print(formula[1,2,3,6,op1,op2])
```

Результат:

```
{(1, 2, 3, 6, '*', '*'), (1, 2, 3, 6, '+', '+')}
```

Заметим, что операции + и * в программе мы обозначили как текст, а не как функции или операции. Запрос написан в стиле исчисления предикатов первого порядка, что означает, что мы не делаем высказываний о высказываниях. Prolog поддерживает только исчисление предикатов первого порядка. В библиотеке DecPy ничто не запрещает нам помещать таблицы внутрь других таблиц, что открывает путь к множеству других логических задач.

6.6. Доказательство теоремы

Язык Prolog не случайно называют языком искусственного интеллекта. С помощью него можно доказывать теоремы. Что может быть интеллектуальнее? Докажем теорему из теории групп на Python с помощью библиотеки DecPy.

Пример взят из книги И.А. Дехтяренко «Декларативное программирование» . Для тех, кто не изучал теорию групп в высшей математике, сделаю пояснение на примере.

Рассмотрим множество целых чисел с бинарной операцией сложения. Для любых чисел `a` имеется противоположное число `-a` такое, что `a+(-a)=0`. Ноль противоположен самому себе и является нейтральным элементом, то есть таким элементом, что для любого `a` выполняется `a+0=a` и `0+a=a`. Система с такими свойствами операций (целые числа с заданной операцией сложения) в математике называется ***группой***. Известно множество полезных групп, помимо чисел, например, группа поворотов плоскости относительно точки или группа симметрий многогранников. Высшая алгебра занимается изучением свойств групп без привязки к конкретному содержимому (числам, поворотам).

Если мы рассмотрим подмножество четных чисел с операцией сложения, мы заметим, что эта операция замкнута относительно подмножества (то есть сумма любых двух четных чисел тоже четное число). Поэтому множество четных чисел также является группой.

Группы не обязательно являются бесконечными.

Докажем простую теорему из теории групп с помощью библиотеки DecPy:

Подмножество группы является подгруппой тогда и только тогда, когда для любых элементов X и Y из этого подмножества результат применения операции X на обратный к Y лежит в этом подмножестве.

Для примера целых чисел с операцией сложения, если теорема верна, то можно утверждать, что четные числа с операцией сложения являются погруппой целых чисел, если доказано, что для любых четных чисел `a` и `b` верно, что `a+(-b)` также четное число.

Докажем задачу с помощью библиотеки DecPy в несколько этапов.

Этап №1. Зададим конечную группу, включив в неё минимально необходимое для доказательства количество элементов. Группа будет абстрактной (не привязанной ни к какой предметной области), то есть элементы будем обозначать буквами, например, `a`, `b`.

Должен существовать нейтральный элемент e:

```
from decpy import table
element=table()
element("e")
print(element)
```

Результат:

```
{'e'}
```

Он должен принадлежать группе. Введем двуместную таблицу «Принадлежность» (на первом месте – элемент, на втором - множество).

```
from decpy import table

element,belong=table(2)

element("e")
print("Элементы:")
print(element)

belong("e","group")
print("Принадлежность:")
print(belong)
```

Результат:

```
Элементы:
{'e'}
Принадлежность:
{('e', 'group')}
```

Дадим определение нейтральности элемента e через некоторую абстрактную операцию: e*x=x и x*e=x.

```
from decpy import var, table
element, operation,belong = table(3)

element("e")
print("Элементы:")
print(element)

x,y,z=var(3)
operation |= (element["e",]**element[y,]**element[z,z==y])
operation |= (element[x,]**element["e",]**element[z,z==x])
print("Таблица операций:")
print(operation)

belong("e","group")
print("Принадлежность:")
print(belong)
```

Результат:

```
Элементы:
{'e'}
Таблица операций:
{('e', 'e', 'e')}
Принадлежность:
{('e', 'group')}
```

Обратите внимание, что в приведенном определении операции `x` является переменной! Хотя мы не определили больше никаких элементов, но когда мы их определим, этот предикат будет выполняться для любых новых элементов.

Библиотека DecPy сделала первое интеллектуальное умозаключение: `e*e=e`!

В теореме имеется формулировка: «для любых элементов `x` и `y`». Это означает, что нам нужны еще два элемента, назовем их `a` и `b`, а также противоположные им `na` и `nb`:

```
element("a")
element("b")
element("na")
element("nb")
```

Все эти элементы должны принадлежать группе. Чтобы их не перечислять, исправим `belong`:

```
belong |= element[x, "group"]
```

Здесь мы фактически берем любой элемент и дописываем к нему еще один атрибут со значением `group`.

```
from decpy import var, table
element,operation,belong=table(3)
x,y,z=var(3)

element("e")
element("a")
element("b")
element("na")
element("nb")
print("Элементы:")
print(element)

operation |= (element["e",]**element[y,]**element[z,z==y])
operation |= (element[x,]**element["e",]**element[z,z==x])
print("Таблица операций:")
print(operation)

belong |= element[x, "group"]
```

```
print("Принадлежность:")
print(belong)
```

Результат:

```
Элементы:
{'nb', 'b', 'a', 'na', 'e'}
Таблица операций:
{('b', 'e', 'b'), ('e', 'nb', 'nb'), ('a', 'e', 'a'), ('e',
'b', 'b'), ('nb', 'e', 'nb'), ('na', 'e', 'na'), ('e', 'e',
'e'), ('e', 'na', 'na'), ('e', 'a', 'a')}
Принадлежность:
{('b', 'group'), ('na', 'group'), ('nb', 'group'), ('a',
'group'), ('e', 'group')}
```

Обратите внимание, как пополнилась таблица операций умножениями на нейтральный элемент всех других элементов.

Надо ввести понятие «противоположность» `invert` для каждого элемента. Напишем противоположные элементы для `a` и `b` (а заодно и для `e`), а наоборот, для `na` и `nb`, пусть DecPy сформулирует автоматически:

```
invert("a","na")
invert("b","nb")
invert("e","e")
invert[x,y]|=invert[y,x]
```

Также нужно пополнить таблицу операций `a*na=e`, `na*a=e`, `b*nb=e`, `nb*b=e`. Запишем, чтобы это пополнение было автоматическим:

```
operation |=
element[x,]**element[y,]**element['e',]**invert[x,y]
```

Обратите внимание, что мы написали эту формулу через переменные `x` и `y`, не перечисляя конкретные элементы.

Приведем программу полностью и посмотрим на результат:

```
from decpy import var, table
element,invert,operation,belong=table(4)
x,y,z=var(3)

element("e")
element("a")
element("b")
element("na")
element("nb")
print("Элементы:")
print(element)
```

```
invert("a","na")
invert("b","nb")
invert("e","e")
invert[x,y]|=invert[y,x]
print("Обратные элементы:")
print(invert)

operation |= (element["e",]**element[y,]**element[z,z==y])
operation |= (element[x,]**element["e",]**element[z,z==x])
operation |=
element[x,]**element[y,]**element['e',]**invert[x,y]
print("Таблица операций:")
print(operation)

belong |= element[x, "group"]
print("Принадлежность:")
print(belong)
```

Результат:

```
Элементы:
{'e', 'na', 'b', 'a', 'nb'}
Обратные элементы:
{('b', 'nb'), ('na', 'a'), ('a', 'na'), ('e', 'e'), ('nb',
'b')}
Таблица операций:
{('a', 'e', 'a'), ('nb', 'e', 'nb'), ('b', 'nb', 'e'),
('b', 'e', 'b'), ('e', 'a', 'a'), ('e', 'b', 'b'), ('na',
'e', 'na'), ('a', 'na', 'e'), ('na', 'a', 'e'), ('e', 'e',
'e'), ('e', 'na', 'na'), ('e', 'nb', 'nb'), ('nb', 'b',
'e')}
Принадлежность:
{('na', 'group'), ('a', 'group'), ('e', 'group'), ('nb',
'group'), ('b', 'group')}
```

По условию теоремы `a` и `b` должны принадлежать подмножеству:

```
belong("a","subgroup")
belong("b","subgroup")
```

Воспользуемся свойством подмножества: все элементы подмножества должны принадлежать также и множеству. Уберем автоматическое добавление всех элементов к группе:

```
belong |= element[x, "group"]
```

и добавим формулу, что все элементы подгруппы входят также в группу:

```
belong |= belong[x, "subgroup", "group"][x,None,y]
```

Мы не стали писать факты, что `a` и `b` принадлежит множеству `group`. Компьютер сам это должен вывести по формуле.

Получаем следующий код программы:

```
from decpy import var, table
element,invert,operation,belong=table(4)
x,y,z=var(3)

element("e")
element("a")
element("b")
element("na")
element("nb")
print("Элементы:")
print(element)

invert("a","na")
invert("b","nb")
invert("e","e")
invert[x,y]|=invert[y,x]
print("Обратные элементы:")
print(invert)

operation |= (element["e",]**element[y,]**element[z,z==y])
operation |= (element[x,]**element["e",]**element[z,z==x])
operation |=
element[x,]**element[y,]**element['e',]**invert[x,y]
print("Таблица операций:")
print(operation)

belong("a","subgroup")
belong("b","subgroup")
belong |= belong[x, "subgroup", "group"][x,None,y]
print("Принадлежность:")
print(belong)
```

Результат:

```
Элементы:
{'nb', 'e', 'a', 'na', 'b'}
Обратные элементы:
{('a', 'na'), ('na', 'a'), ('b', 'nb'), ('nb', 'b'), ('e',
'e')}
Таблица операций:
{('a', 'e', 'a'), ('e', 'na', 'na'), ('na', 'a', 'e'),
('e', 'nb', 'nb'), ('nb', 'b', 'e'), ('b', 'e', 'b'), ('a',
'na', 'e'), ('na', 'e', 'na'), ('e', 'b', 'b'), ('nb', 'e',
'nb'), ('e', 'e', 'e'), ('e', 'a', 'a'), ('b', 'nb', 'e')}
Принадлежность:
{('b', 'group'), ('a', 'group'), ('b', 'subgroup'), ('a',
'subgroup')}
```

По теореме подмножеству subgroup должны также принадлежать и все остальные элементы: e, na, nb. Но теорема-то пока не доказана, поэтому мы можем лишь утверждать, что они принадлежат group:

```
belong("na","group")
belong("nb","group")
belong("e","group")
```

Кроме того, введем элемент m, про который не сказано, что он входит в subgroup:

```
element("m")
belong("m","group")
```

Также по условию нашей теоремы должно существовать произведение a на nb – элемент anb:

```
element("anb")
belong("anb","group")
operation("a","nb","anb")
```

Получаем программу:

```
from decpy import var, table
element,invert,operation,belong=table(4)
x,y,z=var(3)

element("e")
element("a")
element("b")
element("na")
element("nb")
element("m")
element("anb")
print("Элементы:")
print(element)

invert("a","na")
invert("b","nb")
invert("e","e")
invert[x,y]|=invert[y,x]
print("Обратные элементы:")
print(invert)

operation("a","nb","anb")
operation |= (element["e",]**element[y,]**element[z,z==y])
operation |= (element[x,]**element["e",]**element[z,z==x])
operation |=
element[x,]**element[y,]**element['e',]**invert[x,y]
```

```
print("Таблица операций:")
print(operation)

belong("a","subgroup")
belong("b","subgroup")
belong("na","group")
belong("nb","group")
belong("e","group")
belong("m","group")
belong("anb","group")
belong |= belong[x, "subgroup", "group"][x,None,y]
print("Принадлежность:")
print(belong)
```

Результат:

```
Элементы:

{'nb', 'm', 'a', 'e', 'b', 'na', 'anb'}
Обратные элементы:
{('na', 'a'), ('b', 'nb'), ('e', 'e'), ('nb', 'b'), ('a',
'na')}
Таблица операций:
{('nb', 'b', 'e'), ('b', 'e', 'b'), ('nb', 'e', 'nb'),
('e', 'm', 'm'), ('na', 'a', 'e'), ('e', 'na', 'na'), ('a',
'nb', 'anb'), ('e', 'b', 'b'), ('e', 'nb', 'nb'), ('m',
'e', 'm'), ('b', 'nb', 'e'), ('na', 'e', 'na'), ('anb',
'e', 'anb'), ('e', 'e', 'e'), ('e', 'anb', 'anb'), ('a',
'e', 'a'), ('a', 'na', 'e'), ('e', 'a', 'a')}
Принадлежность:
{('anb', 'group'), ('b', 'subgroup'), ('na', 'group'),
('nb', 'group'), ('a', 'subgroup'), ('b', 'group'), ('m',
'group'), ('a', 'group'), ('e', 'group')}
```

Этап 2. Формулировка теоремы.

Рассмотрим заключение теоремы:

«…для любых элементов X и Y из этого подмножества результат применения операции X на обратный к Y лежит в этом подмножестве.»

То есть `anb` должен принадлежать `subgroup`. Надо сформулировать это с помощью библиотеки DecPy.

Принадлежность любых элементов `x` и `y` к подмножеству:

```
belong[x,"subgroup"]**belong[y,"subgroup"]
```

У `y` надо взять обратный элемент:

```
Belong[x,"subgroup"]**belong[y,"subgroup"]**invert[y,ny]
```

«*Результат лежит в подмножестве*». Обозначим результат переменной `z`.

```
Belong[x,"subgroup"]**belong[y,"subgroup"]**invert[y,ny]**o
peration[x,ny,z]
```

и напишем, что он принадлежит подмножеству:

```
belong |=
belong[x,"subgroup"]**belong[y,"subgroup"]**invert[y,ny]**o
peration[x,ny,z][None,None,None,None,z,'subgroup']
```

Добавим эту формулу к программе:

```
from decpy import var, table
element,invert,operation,belong=table(4)
x,y,z,ny=var(4)

element("e")
element("a")
element("b")
element("na")
element("nb")
element("m")
element("anb")
print("Элементы:")
print(element)

invert("a","na")
invert("b","nb")
invert("e","e")
invert[x,y]|=invert[y,x]
print("Обратные элементы:")
print(invert)

operation("a","nb","anb")
operation |= (element["e",]**element[y,]**element[z,z==y])
operation |= (element[x,]**element["e",]**element[z,z==x])
operation |=
element[x,]**element[y,]**element['e',]**invert[x,y]
print("Таблица операций:")
print(operation)

belong("a","subgroup")
belong("b","subgroup")
belong("na","group")
belong("nb","group")
belong("e","group")
belong("m","group")
belong("anb","group")
belong |= belong[x, "subgroup", "group"][x,None,y]
```

```
belong |=
(belong[x,'subgroup']**belong[y,'subgroup']**invert[y,ny]**
operation[x,ny,z])[None,None,None,None,z,'subgroup']
print("Принадлежность:")
print(belong)
```

Результат:

```
Элементы:
{'na', 'a', 'anb', 'e', 'b', 'm', 'nb'}
Обратные элементы:
{('a', 'na'), ('nb', 'b'), ('b', 'nb'), ('e', 'e'), ('na',
'a')}
Таблица операций:
{('a', 'e', 'a'), ('e', 'nb', 'nb'), ('na', 'e', 'na'),
('e', 'm', 'm'), ('nb', 'b', 'e'), ('na', 'a', 'e'), ('e',
'e', 'e'), ('e', 'b', 'b'), ('e', 'a', 'a'), ('m', 'e',
'm'), ('a', 'na', 'e'), ('anb', 'e', 'anb'), ('a', 'nb',
'anb'), ('b', 'e', 'b'), ('e', 'na', 'na'), ('e', 'anb',
'anb'), ('b', 'nb', 'e'), ('nb', 'e', 'nb')}
Принадлежность:
{('na', 'group'), ('m', 'group'), ('anb', 'subgroup'),
('nb', 'subgroup'), ('anb', 'group'), ('nb', 'group'),
('b', 'group'), ('e', 'subgroup'), ('e', 'group'), ('b',
'subgroup'), ('a', 'subgroup'), ('na', 'subgroup'), ('a',
'group')}
```

Этап 3. Проверка правильности теоремы.

Теорема утверждает, что «*подмножество группы является подгруппой»* в математическом смысле. `Subgroup` является группой в математическом смысле, если выполняются требования, предъявляемые группе:

Для любых чисел `a` имеется противоположное число `−a` такое, что `a+(-a)=0`. Ноль противоположен самому себе и является нейтральным элементом, то есть таким элементом, что для любого `a` выполняется `a+0=a` и `0+a=a`.

Здесь есть несколько утверждений. Для каждого из них сформулируем вопрос, который зададим системе при выполнении программы. Утвердительные ответы на каждый вопрос будут означать, что теорема верна.

1) В группе есть нейтральный элемент. Очевидно, что это элемент `e`. Пока у нас введен факт, что `e` принадлежит `group`. Проверим, принадлежит ли он `subgroup`:

```
print(belong["e","subgroup"])
```

2) Принадлежит ли элемент, противоположный элементу `a`, группе `subgroup`:

```
print(belong["na","subgroup"])
```

3) Произведение a на любой другой элемент subgroup должно принадлежать к subgroup:

Выведем все элементы подгруппы:

```
print(belong[x,"subgroup"])
```

Пусть y – результат произведения элемента a на x:

```
belong[x,"subgroup"]**operation["a",x,y]
```

Элемент y должен принадлежать подгруппе:

```
belong[x,"subgroup"]**operation["a",x,y]**
belong[y,"subgroup"]
```

Выведем последние два запроса на экран. Приведем программу полностью:

```
from decpy import var, table
element,invert,operation,belong=table(4)
x,y,z,ny=var(4)

element("e")
element("a")
element("b")
element("na")
element("nb")
element("m")
element("anb")
print("Элементы:")
print(element)

invert("a","na")
invert("b","nb")
invert("e","e")
invert[x,y]|=invert[y,x]
print("Обратные элементы:")
print(invert)

operation("a","nb","anb")
operation |= (element["e",]**element[y,]**element[z,z==y])
operation |= (element[x,]**element["e",]**element[z,z==x])
operation |=
element[x,]**element[y,]**element['e',]**invert[x,y]
print("Таблица операций:")
print(operation)

belong("a","subgroup")
belong("b","subgroup")
belong("na","group")
```

```
belong("nb","group")
belong("e","group")
belong("m","group")
belong("anb","group")
belong |= belong[x, "subgroup", "group"][x,None,y]
belong |=
(belong[x,'subgroup']**belong[y,'subgroup']**invert[y,ny]**
operation[x,ny,z])[None,None,None,None,z,'subgroup']
print("Принадлежность:")
print(belong)
print("Нейтральный элемент принадлежит подгруппе?")
print(belong["e","subgroup"])
print("Принадлежит ли элемент, противоположный к a
подгруппе?")
print(belong["na","subgroup"])
print("Все элементы подгруппы:")
print(belong[x,"subgroup"])
print("Результаты умножения a на элементы подгруппы:")
print((belong[x,"subgroup"]**operation["a",x,y]))
print("Принадлежит ли результат умножения a на элементы
подгруппы подгруппе:")
print(belong[x,"subgroup"]**operation["a",x,y]**
belong[y,"subgroup"])
```

Результат:

```
Элементы:
{'anb', 'm', 'a', 'na', 'nb', 'b', 'e'}
Обратные элементы:
{('a', 'na'), ('nb', 'b'), ('b', 'nb'), ('na', 'a'), ('e',
'e')}
Таблица операций:
{('e', 'a', 'a'), ('b', 'nb', 'e'), ('nb', 'b', 'e'), ('e',
'nb', 'nb'), ('a', 'nb', 'anb'), ('e', 'm', 'm'), ('a',
'e', 'a'), ('anb', 'e', 'anb'), ('e', 'b', 'b'), ('b', 'e',
'b'), ('nb', 'e', 'nb'), ('m', 'e', 'm'), ('na', 'a', 'e'),
('na', 'e', 'na'), ('a', 'na', 'e'), ('e', 'na', 'na'),
('e', 'anb', 'anb'), ('e', 'e', 'e')}
Принадлежность:
{('b', 'group'), ('anb', 'group'), ('na', 'subgroup'),
('a', 'subgroup'), ('na', 'group'), ('nb', 'subgroup'),
('b', 'subgroup'), ('nb', 'group'), ('e', 'subgroup'),
('e', 'group'), ('a', 'group'), ('m', 'group'), ('anb',
'subgroup')}
Нейтральный элемент принадлежит подгруппе?
{('e', 'subgroup')}
Принадлежит ли элемент, противоположный к a подгруппе?
{('na', 'subgroup')}
Все элементы подгруппы:
```

```
{('na', 'subgroup'), ('a', 'subgroup'), ('nb', 'subgroup'),
('e', 'subgroup'), ('b', 'subgroup'), ('anb', 'subgroup')}
Результаты умножения a на элементы подгруппы:
{('e', 'subgroup', 'a', 'a'), ('nb', 'subgroup', 'a',
'anb'), ('na', 'subgroup', 'a', 'e')}
Принадлежит ли результат умножения a на элементы подгруппы
подгруппе:
{('e', 'subgroup', 'a', 'a'), ('nb', 'subgroup', 'a',
'anb'), ('na', 'subgroup', 'a', 'e')}
```

Из результата видно, что DecPy нашла элементы `e` и `na` в подгруппе. Кроме того, последние два множества полностью совпадают. Это значит, что результаты умножения `a` на элементы подгруппы все находятся в подгруппе.

Может возникнуть вопрос о том, что мы задали вопросы только для элемента подмножества `a`. Но ведь мы и сформулировали теорему без какой-либо конкретики! То есть под `a` может пониматься любой объект реальной алгебраической системы. А это значит, что нам не нужно проверять верность теоремы для `b`. Это довольно распространенный прием при доказательстве теорем в математике.

Заметим, что хотя поиск-доказательство теоремы DecPy выполнила самостоятельно, но «человека пока еще никто не отменял». Формулировка теоремы в терминах логического программирования – это сама по себе трудная задача, требующая от программиста интеллектуальных усилий!

Этой важной задачей, доказательством теоремы, мы заканчиваем книгу.

Выводы

В этом разделе мы решили с помощью библиотеки DecPy задачи, которые обычно решают на языке Prolog в области искусственного интеллекта. И, как и решения на Prolog, они имеют такие же достоинства и недостатки. Часть из этих задач пока являются «программистской игрушкой» (поиск операций по числам, факториал), но могут иметь в будущем теоретическое и практическое значение. Некоторые решения работают медленно (схемотехнические задачи и функции через запросы), но возможно их ускорение. Другие же, например, доказательство теорем, являются важной задачей из области искусственного интеллекта.

Заключение

Итак, с помощью библиотеки DecPy мы научились писать запросы к спискам, множествам, составным коллекциям, функциям и классам, как будто это таблицы из баз данных. Причем эти запросы можно писать в стиле как SQL, так и QBE. В том числе эти запросы могут быть рекурсивными. Запросы в стиле Prolog позволили создавать на основе имеющихся исходных коллекций (классов) новые понятия. По сути, это дополнение объектно-ориентированного программирования, в котором традиционный механизм образования новых понятий – это наследование. В сочетании друг с другом и такими стандартными средствами Python, как срезы, прямая и обратная адресация, мы научились просто и лаконично решать задачи на графах. Также на Python стало возможно доказывать теоремы.

Все разработанные средства органично вписались в язык Python, развивая имеющиеся у него механизмы в виде оператора квадратных скобок. Теперь декларативное и логическое программирование заняло свое место среди других парадигм языка Python.

Приложение. Обзор библиотеки DecPy для профессионалов

В приложении для профессионалов, владеющих не только Python, но и SQL, на примерах дается обзор возможностей библиотеки DecPy. Желательно также знание на минимальном уровне Prolog, понимание реляционной алгебры, исчислений на кортежах, доменах, понимание предикатов первого порядка; либо большой практический опыт в программировании, позволяющий интуитивно «схватывать суть кода». Если вы не относитесь к таким программистам, читайте полный учебник по DecPy с самого начала.

П1. Переменные var – основа ленивых вычислений

Объявление переменных:

```
a=var()
```

- это основа ***ленивых*** (отложенных) ***вычислений***.

Возможно объявление сразу нескольких переменных:

```
a,b=var(2)
```

Вот пример ленивых вычислений:

```
a,b = var(2)
pithagor=(a**2+b**2)**0.5
a(3)
b(4)
print(pithagor)
a(5)
b(12)
print(pithagor)
```

Результат:

```
5.0
13.0
```

Здесь инициализация переменных производится с помощью указания значения в круглых скобках:

```
a(3)
```

Если проинициализировать переменную коллекцией, то мы будем иметь дело с ***ленивой коллекцией***:

```
from decpy import var
L=[1,2,3,4,5]
M=var(L)
print(M)
L.append(6)
print(M)
```

Результат:

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 6]
```

К ленивым коллекциям возможны запросы (выборки, селекция) в стиле языков SQL, QBE и Prolog (см. пункт П3).

Переменные с помощью операций арифметики соединяются в ***ленивые формулы***, которые могут использоваться как функции:

```
from decpy import var

a,b = var(2)
pithagor=(a**2+b**2)**0.5
print(pithagor(3,4))
print(pithagor(5,12))
```

П2. Ленивые функции lazyfun

В ленивые формулы можно встраивать библиотечные функции, которые нужно предварительно преобразовать с помощью `lazyfun`:

```
from math import sqrt
from decpy import var,lazyfun

sqrt=lazyfun(sqrt)

a,b = var(2)
pithagor=sqrt(a**2+b**2)
print(pithagor(3,4))
```

`lazyfun` также можно использовать как декоратор при объявлении собственных функций в программе.

```
from decpy import var,lazyfun

@lazyfun
def sqrt(a):
    return a**0.5

a,b = var(2)
pithagor=sqrt(a**2+b**2)
print(pithagor(3,4))
```

П3. Оператор […] – основа запросов

Условия выборок из ленивых коллекций пишутся в квадратных скобках после коллекции – так получаются ***ленивые запросы***:

```
from decpy import var

el=var()
L=[10,20,30,40,50]
M=var(L)[el>15]
print(M)
L.append(60)
print(M)
```

Результат:

```
[20, 30, 40, 50]
[20, 30, 40, 50, 60]
```

Здесь ленивая переменная `el` используется внутри запроса, и её смысл – перебор элементов коллекции (переменная – кортеж в терминах ***исчисления на кортежах***).

Возможна последовательность условий в виде серии квадратных скобок:

```
from decpy import var

el=var()
L=[10,20,30,40,50]
P=var(L)[el>15][el<45]
print(L)
print(P)
```

Результат:

```
[10, 20, 30, 40, 50]
[20, 30, 40]
```

Условия можно объединять внутри одних квадратных скобок с помощью операторов «`&`» (логическое «и») и «`|`» логическое «или»):

```
from decpy import var

el=var()
L=[10,20,30,40,50]
P=var(L)[(el>15) & (el<45)]
print(L)
print(P)
```

П4. Запросы к классам queryclass

Запросы можно делать не только к коллекциям, инициализированным с помощью var по типу:

```
L=[1,2,3,4,5]
M=var(L)[el>2]
```

, но и к множеству всех экземпляров класса. Для этого класс надо оформить с помощью декоратора `queryclass`. В следующем примере создаются точки на плоскости – экземпляры класса `point`. Затем выводятся точки, попадающие в квадрат размерами 5x5:

```
from decpy import var,queryclass

@queryclass
class point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
    def __repr__(self):
        return "("+str(self.x)+","+str(self.y)+")"
    def abs(self):
        return (self.x**2+self.y**2)**0.5

point(6,0)
point(1,7)
point(0,4)
point(4,4)
point(1,3)
point(6,6)
point(3,4)
print(point)
el=var()
M=point[(el.x<=5) & (el.y<=5)]
print(M)
```

Результат:

```
[(6,0), (1,7), (0,4), (4,4), (1,3), (6,6), (3,4)]
[(0,4), (4,4), (1,3), (3,4)]
```

Заметим, что при объявлении переменной

```
el=var()
```

библиотека не знает, что у переменной `el` есть атрибуты `x` и `y`. Здесь мы имеем дело с ***ленивыми атрибутами***. Также библиотека DecPy поддерживает ***ленивые индексы***. Ленивые индексы и атрибуты могут комбинироваться различными способами, например:

```
A,B,C,D=var(3)
…
print(A[0].x[1])
print(B.x[1].y)
print(C[1][2].x.y)
print(D.x.y[1][2])
```

П5. Таблица table

Если класс не предусматривает методов, а его экземпляры фактически объединяются в таблицу, в которой имена колонок – это названия атрибутов, то вместо класса с декоратором `queryclass` проще использовать `table` из

библиотеки DecPy. Вот пример с фильтрацией точек из предыдущего пункта с помощью `table`:

```
from decpy import var, table

point=table("x","y")
point(6,0)
point(1,7)
point(0,4)
point(4,4)
point(1,3)
point(6,6)
point(3,4)
print(point)
el=var()
M=point[(el.x<=5) & (el.y<=5)]
print(M)
```

Код стал лаконичнее! Но `table` предназначена не только для сокращения кода. Это мощное средство для написания запросов в стиле Prolog (см. пункт П10)

П6. Запросы в стиле исчисления на кортежах и исчисления на доменах

Приведу несколько запросов для таблицы точек на плоскости (уверен, что читатель легко поймет смысл запросов):

```
from decpy import var,table

point=table("x","y")
point(6,0)
point(1,7)
point(0,4)
point(4,4)
point(1,3)
point(6,6)
point(3,4)
print(point)
el=var()
print(point[el.x])
print(point[el.y])
print(point[(el.x<=5) & (el.y<=5)])
print(point[(el.x<=5) | (el.y<=5)])
print(point[el.x**2+el.y**2<=25])
```

Результат:

```
[(6, 0), (1, 7), (0, 4), (4, 4), (1, 3), (6, 6), (3, 4)]
{0, 1, 3, 4, 6}
{0, 3, 4, 6, 7}
{(4, 4), (1, 3), (3, 4), (0, 4)}
{(4, 4), (0, 4), (3, 4), (1, 7), (6, 0), (1, 3)}
{(1, 3), (3, 4), (0, 4)}
```

Здесь смысл использования переменной `el` – это перебрать все элементы коллекции (`el` - ***переменная-кортеж*** в терминах математического аппарата ***исчисления на кортежах***). Многократно писать `el` утомительно. Можно использовать две переменные `x` и `y` и написать эти же самые запросы более изящно:

```
from decpy import var

L=var([(6,0),(1,7),(0,4),(4,4),(1,3),(6,6),(3,4)])
x,y,d=var(3)
print(L)
print(L[x, None])
print(L[None, y])
print(L[x<=5, y<=5])
print(L[x, y, (x<=5) | (y<=5)])
print(L[x, y, x**2+y**2<=25])
```

Здесь `x` и `y` – ***переменные-домены*** в терминах ***исчисления на доменах***.

Если на исчислении на кортежах основан язык SQL, то на исчислении на доменах – язык QBE. Библиотека DecPy позволяет комбинировать оба подхода (а также запросы в стиле Prolog), выбирая в каждом случае наиболее удобный вариант.

П7. Запросы к функциям queryfun, ленивый range

Задачи, связанные с поиском по таблицам, составленным на основе вычислений функций, не такая уж и редкость. Подобно тому, как декоратор `queryclass` автоматически дополняет класс его экстентом (коллекцией для хранения его экземпляров), в библиотеке DecPy есть декоратор `queryfun`, сохраняющий аргументы функции вместе с результатом её выполнения.

Сформируем таблицу умножения с помощью `queryfun` и напишем разные запросы к этой таблице, в том числе – разложение числа 12 на два множителя:

```
from decpy import var, lazyrange, queryfun

@queryfun
def mult(x,y):
    return x*y

mult.init(lazyrange(7),lazyrange(7))
print(mult(3,4))
x,y,z=var(3)
print(mult[3,4,z])
print(mult[3,y,12])
print(mult[x,y,12])
print(mult[3,4,12])
print(mult[3,4,15])
```

Результат:

```
12
{3, 4, 12}
{(3, 4, 12)}
{(4, 3, 12), (2, 6, 12), (3, 4, 12), (6, 2, 12)}
True
False
```

Здесь `init` заполняет таблицу функции, инициализируя аргументы областью определения функции.

`Lazyrange` – это ***ленивый генератор диапазонов*** `range` с расширенными, по сравнению со стандартным `range` из Python, возможностями. Так, с его помощью можно генерировать не только арифметические, но и геометрические прогрессии, использовать дробный шаг и даже получать рекуррентные последовательности, например, числа Фибоначчи (см. главу 6.2.).

Заметим, что `mult` продолжает работать и как обычная функция, и как таблица для запросов.

П8. Возможности SQL в DecPy

DecPy содержит аналогичный SQL набор вспомогательных операторов для удаления дубликатов, группировок, сортировок; а также включает кванторы и операторы для объединения однотипных запросов. Реализованы они как ***ленивые методы***. Приведем некоторые примеры:

Пример 1. Удаление дубликатов:

```
from decpy import var

el=var()
L=var([(1,2,1),(10,20,30,10,20),(100,100,100)])
M=L[el.distinct()]
print(L)
print(M)
```

Результат:

```
[(1, 2, 1), (10, 20, 30, 10, 20), (100, 100, 100)]
[(1, 2), (10, 20, 30), (100,)]
```

Пример 2. Сортировка элементов**:**

```
from decpy import var

el=var()
L=var([(1,3,2),(2,1,4,3),(30,50,10,20)])
M=L[el.sorted()]
print(L)
print(M)
```

Результат:

```
[(1, 3, 2), (2, 1, 4, 3), (30, 50, 10, 20)]
[[1, 2, 3], [1, 2, 3, 4], [10, 20, 30, 50]]
```

Пример 3. Подсчет длины элементов:

```
from decpy import var

el=var()
L=var([(10,20),(30,40,50),(60,70,80,90)])
M=L[el.len()]
print(L)
print(M)
```

Результат:

```
[(10, 20), (30, 40, 50), (60, 70, 80, 90)]
[2, 3, 4]
```

Аналогично с помощью `min`, `max`, `avg` и `sum` можно посчитать минимальный, максимальный элемент, среднее арифметическое и сумму.

Пример 4. Группировка; запрос с исчислением на доменах:

```
from decpy import var

x,y=var(2)
L=var([("A",10),("A",20),("A",30),("B",15),("B",55),("C",40
)])
M=L[x.group(),y]
print(L)
print(M)
```

Результат:

```
[('A', 10), ('A', 20), ('A', 30), ('B', 15), ('B', 55),
('C', 40)]
{('A', (10, 30, 20)), ('B', (15, 55)), ('C', (40,))}
```

Пример 5. Группировка с подсчетом сумм; запрос с исчислением на доменах:

```
from decpy import var

x,y=var(2)
L=var([("A",10),("A",20),("A",30),("B",15),("B",55),("C",40
)])
M=L[x.group(),y.sum()]
print(L)
print(M)
```

Результат:

```
[('A', 10), ('A', 20), ('A', 30), ('B', 15), ('B', 55),
('C', 40)]
{('B', 70), ('C', 40), ('A', 60)}
```

Пример 6. Комбинированнный – из предыдущего примера выведем максимальный элемент по посчитанной сумме:

```
from decpy import var

x,y,el=var(3)
L=var([("A",10),("A",20),("A",30),("B",15),("B",55),("C",40
)])
M=L[x.group(),y.sum()].max(el[1])
print(L)
print(M)
```

Результат:

```
[('A', 10), ('A', 20), ('A', 30), ('B', 15), ('B', 55),
('C', 40)]
('B', 70)
```

Объединяются результаты запросов с помощью операторов работы с множествами Python: «&», «|», «–», «^» по следующей схеме:

```
M = L[…]
N = L[…]
P = M | N
```

П9. Операторы декартова произведения «*» и «**» - соединение коллекций

Типичными запросами в базах данных являются запросы к нескольким таблицам. В их основе лежит ***декартово произведение*** с последующей ***селекцией***. Например, нужно соединить две коллекции так, чтобы нулевые атрибуты соединяемых элементов совпадали. Для декартова произведения в DecPy используются два оператора. Первый – «*»:

```
from decpy import var

l,m=var(2)
L=var({("A","aa"),("B","bb")})
M=var({("A",1),("A",2),("B",30)})
P=(L*M)[l,m,l[0]==m[0]]
print(P)
```

Результат:

```
{(('A', 'aa'), ('A', 1)), (('A', 'aa'), ('A', 2)), (('B',
'bb'), ('B', 30))}
```

Как видно, оператор «*» сохраняет внутреннюю структуру исходных элементов, порождая составной элемент, например:

```
(('A', 'aa'), ('A', 1))
```

Более тесное соединение, разрушающее внутреннюю структуру элементов, обеспечивает оператор «**». Он удобен для запросов в стиле исчисления на доменах:

```
from decpy import var

x,y,z,u=var(4)
L=var({("A","aa"),("B","bb")})
M=var({("A",1),("A",2),("B",30)})
P=(L**M)[x,y,z,u,x==z]
print(P)
```

Результат:

```
{('A', 'aa', 'A', 1), ('A', 'aa', 'A', 2), ('B', 'bb', 'B',
30)}
```

Теперь исходные элементы соединены в один, но у них появились дублирующие колонки. Убрать дублирующие колонки можно изящным способом:

```
from decpy import var

x,y,z,u=var(4)
L=var({("A","aa"),("B","bb")})
M=var({("A",1),("A",2),("B",30)})
P=(L**M)[x,y,x,u]
print(P)
```

Результат:

```
{('A', 'aa', 1), ('A', 'aa', 2), ('B', 'bb', 30)}
```

П10. Оператор определения «|=». Prolog-подобные запросы и рекурсивные запросы

Продолжим изучать последний пример. В случае соединения более чем двух таблиц можно запутаться в переменных-доменах. Библиотека DecPy позволяет писать переменные рядом с теми таблицами, к которым они относятся. Того же самого результата можно добиться с помощью следующего кода:

```
from decpy import var, table

x,y,z=var(3)
L=var({("A","aa"),("B","bb")})
M=var({("A",1),("A",2),("B",30)})
P=table()
P=L[x,y]**M[x,z]
print(P)
```

Обратите внимание, что `P` пришлось объявить как `table`.

Если связующая колонка `x` в ответе не нужна, то можно ее удалить из `P`, просто прописав рядом с `P` названия нужных колонок, входящих в ответ:

```
from decpy import var, table

x,y,z=var(3)
L=var({("A","aa"),("B","bb")})
M=var({("A",1),("A",2),("B",30)})
P=table()
P[y,z] |= L[x,y]**M[x,z]
print(P)
```

Результат:

```
{('aa', 1), ('aa', 2), ('bb', 30)}
```

Обратите внимание, что оператор присваивания «=» у переменной P мы заменили на оператор «|=». Здесь мы от исчисления на доменах перешли к родственному ему исчислению предикатов первого порядка, на котором основывается язык искусственного интеллекта Prolog.

Строчка кода:

```
P[y,z] |= L[x,y]**M[x,z]
```

- это вполне прологовская запись (только квадратным скобкам в Prolog соответствуют круглые, а оператору «|=» в Prolog «:-»).

В полной мере красота Prolog-подобных запросов будет видна в следующей главе.

П11. Рекурсивные запросы

Приведем пример Prolog-подобных запросов, в которых проявится мощь библиотеки DecPy. Решим задачу, связанную с родословным древом. Пусть между людьми существуют отношения ребенок / родитель. Задать их можно различными способами, выберем самый простой:

```
from decpy import table
parent=table()
parent("Вася","Иван") #Вася сын Ивана
parent("Иван","Григорий") #Иван сын Григория
print(parent)
```

Результат:

```
{('Вася', 'Иван'), ('Иван', 'Григорий')}
```

Напишем определение, кто такой дедушка – это родитель родителя. Формализуем это определение через переменные: y является дедушкой для x, если родителем у x является человек z, причем z является ребенком y:

```
from decpy import var,table
x,y,z=var(3)
parent,grandparent=table(2)
grandparent[x,y]|=parent[x,z]**parent[z,y]
parent("Вася","Иван")  #Вася сын Ивана
parent("Иван","Григорий")  #Иван сын Григория
print(grandparent)
```

Результат:

```
{('Вася', 'Григорий')}
```

Введем понятие предок – это родитель или дедушка. Выведем предков Васи:

```
from decpy import var,table
x,y,z=var(3)
parent,grandparent,ancestor=table(3)
grandparent[x,y] |= parent[x,z]**parent[z,y]
ancestor |= parent | grandparent
parent("Вася","Иван")  #Вася сын Ивана
parent("Иван","Григорий")  #Иван сын Григория
print(ancestor["Вася",x])
```

Результат:

```
{('Вася', 'Иван'), ('Вася', 'Григорий')}
```

Если определение дедушки не нужно, мы можем его удалить, скопировав код в определение предка:

```
from decpy import var,table
x,y,z=var(3)
parent,ancestor=table(2)
ancestor[x,y] |= parent[x,y] | parent[x,z]**parent[z,y]
parent("Вася","Иван")  #Вася сын Ивана
parent("Иван","Григорий")  #Иван сын Григория
print(ancestor["Вася",x])
```

Введем в родословное древо еще один уровень – сделаем Григория сыном Глеба:

```
from decpy import var,table
x,y,z=var(3)
parent,ancestor=table(2)
ancestor[x,y] |= parent[x,y] | parent[x,z]**parent[z,y]
parent("Вася","Иван")  #Вася сын Ивана
parent("Иван","Григорий")  #Иван сын Григория
parent("Григорий","Глеб")  #Григорий сын Глеба
print(ancestor["Вася",x])
```

Результат не изменился:

```
{('Вася', 'Иван'), ('Вася', 'Григорий')}
```

Глеб является прадедушкой Васи, но не входит в число предков, так как Васю и Глеба по цепочке родства связывает не один человек (переменная `z` в программе, а два человека). Бесконечно увеличивать цепочку в определении предка мы не можем, но есть изящное решение. Пусть `z` является родителем для `x`. Тогда кем является `y` для `z` в общем случае? Предком! Заменим второе вхождение `parent` в запрос на `ancestor`:

```
from decpy import var,table
x,y,z=var(3)
parent,ancestor=table(2)
ancestor[x,y] |= parent[x,y] | parent[x,z]**ancestor[z,y]
parent("Вася","Иван")  #Вася сын Ивана
parent("Иван","Григорий")  #Иван сын Григория
parent("Григорий","Глеб")  #Григорий сын Глеба
print(ancestor["Вася",x])
```

Результат:

```
{('Вася', 'Глеб'), ('Вася', 'Григорий'), ('Вася', 'Иван')}
```

Теперь все предки Васи найдены.

Рекурсивные запросы – это мощное средство, и в Prolog (а теперь и в Python) они реализованы наиболее красивым и лаконичным способом, если сравнивать с другими декларативными языками. Больше красивых рекурсивных запросов к родословному древу вы найдете в разделе 4.

Но на полную мощь возможности библиотеки DecPy разворачиваются тогда, когда мы применяем их совместно (рекурсии, запросы в стиле SQL, QBE, Prolog и стандартные средства Python: прямую и обратную индексации, срезы). Примером является задача о самолетных рейсах с любым количеством пересадок, описанная в 5 разделе.

Библиотека DecPy начинается с ленивых вычислений и на их основе реализует три вида декларативного программирования, которые можно комбинировать. Теперь программист может полениться – с помощью библиотеки DecPy код стал короче и понятнее.

Предметный указатель

Редактор: Хрущева Е.Н.

Добряк П.В. Библиотека DecPy. Декларативное программирование на Python. – Электронное издание. Екатеринбург, 2025. – 144 с.: ил.