PySIMPLE Documentation

リリース *1.6.0*

nuopt-support@ml.msi.co.jp

2025 年 02 月 25 日

目 次

第1章　はじめに

1.1 PySIMPLE について

PySIMPLE (pronounced 'pie simple') は Nuorium Optimizer の Python インタフェースです.

従来より Nuorium Optimizer 付属の モデリング言語 C++SIMPLE の文法をできるだけ Python 環境に取り入れることにより，Nuorium Optimizer をご利用いただいていたユーザは違和感なく，新たにご利用いただくユーザにはモデリング言語 C++SIMPLE と共にご利用いただくことができます.

Nuorium Optimizer は幅広い数理最適化問題に対して豊富な解法を提供する数理最適化パッケージです. PySIMPLE ではこのうち (混合整数) 線形計画問題，凸 (混合整数) 二次計画問題，二次制約付き二次計画問題，および制約充足問題に対して Nuorium Optimizer ソルバーへの接続インターフェースを提供します.

1.2 動作環境

Nuorium Optimizer V27 に同梱されている PySIMPLE 1.6.0 は以下の環境で動作します.

- OS
  - Windows : Windows 10, 11, これら OS の Server 版
  - macOS : 13(Ventura), 14(Sonoma), 15(Sequoia)
  - Linux : CentOS Stream 9(RHEL 9), Ubuntu 22.04 LTS, Ubuntu 24.04 LTS, Amazon Linux 2023
- Python
  - 3.10, 3.11, 3.12, 3.13 (いずれも 64bit のみ)
  - 依存モジュール
    - scipy (>=1.8.1)
    - numpy (>=1.22.4 from scipy>=1.8.1)
- Nuorium Optimizer : V27 開発版のライセンスが有効である環境

過去のバージョンの動作環境は Nuorium Optimizer 対応機種／ OS#PySIMPLE 動作環境をご確認ください.

1.3 インストール

PySIMPLE をご利用いただくに先立って Nuorium Optimizer と Python をインストールしておく必要があります.

Nuorium Optimizer のインストール (Windows 版) については下記よりインストールガイドをご覧ください.

- https://www.msi.co.jp/solution/nuopt/docs/index.html

Python のインストール (Windows 版) については下記をご覧ください.

- https://docs.python.org/ja/3/using/windows.html

PySIMPLE のインストールは利用する Python の pip にて行います．以下，インストールの手順になります.

PySIMPLE のインストール先は，利用する Python のインストール先の「lib\site-packages」フォルダになります．そのため，インストールするユーザは本フォルダに対して書き込み権限が必要となります．インストール作業を実行する際は，本フォルダに対して書き込み権限があるユーザでログオンしてください.

1. PySIMPLE のインストール用ファイルを用意する

   Windows 版の PySIMPLE のインストールファイル用フォルダは

   %DRIVE%\pysimple

   です．%DRIVE% は Nuorium Optimizer のインストールメディアをマウントした ISO のドライブを表します．macOS 版，Linux 版の PySIMPLE のインストールファイル用ディレクトリは

   (Nuorium Optimizer のインストール先のディレクトリ)/nuopt/pysimple/

   です．インストール用のファイル (wheel ファイル) の名前は以下となります.

   - Windows 版 : pysimple-1.6.0-cp3x-cp3x-win_amd64.whl
   - macOS 版 : pysimple-1.6.0-cp3x-cp3x-macosx_15_0_x86_64.whl
   - Linux 版 : pysimple-1.6.0-cp3x-cp3x-linux_x86_64.whl

   cp3x はご利用になる Python のバージョンを表す cp310, cp311, cp312, cp313 のいずれかになります.

2. コンソールを起動する

   Windows 版の場合は，コマンドプロンプトを起動してください．Python のインストール先の書き込み権限がない場合は，コマンドプロンプトを起動する際に右クリックメニューの「管理者として実行」から起動する必要があります．Anaconda をご利用の場合は Anaconda Prompt を起動してください．macOS，Linux 版の場合は端末を起動してください.

   python コマンドにパスが通っていれば以下のようにしてご利用環境の Python のバージョンを確かめられます．macOS，Linux 版の場合は python コマンドの代わりに python3 コマンドを使用してください．:

```
$ python --version
Python 3.13.2
```

   上記のようにバージョンを確認したあと，適切な wheel ファイルを選択してください．上記の例では Windows 版で Python 3.13 のため，pysimple-1.6.0-cp313-cp313-win_amd64.whl を選びます.

3. wheel ファイルを使ってインストールする

   コンソール上で pysimple フォルダに移動し選択した wheel ファイルを用いてインストールします．以下は Windows 版の例です．：

```
$ cd /d %DRIVE%\pysimple
$ python -m pip install pysimple-1.6.0-cp3x-cp3x-win_amd64.whl
Processing c:\users\uname\desktop\pysimple151\pysimple-1.6.0-cp3x-cp3x-win_amd64.
↪whl
Requirement already satisfied: scipy>=1.6.3 in c:\users\uname\appdata\local\
↪programs\python\python3x\lib\site-packages (from pysimple==1.6.0) (1.13.1)
Requirement already satisfied: numpy<2.3,>=1.22.4 in c:\users\uname\appdata\
↪local\programs\python\python3x\lib\site-packages (from scipy>=1.6.3->
↪pysimple==1.6.0) (2.0.0)
Installing collected packages: pysimple
Successfully installed pysimple-1.6.0
```

   このように最後に「Successfully installed pysimple-1.6.0」と表示されればインストールは成功です．また，numpy，scipy がインストールされていない，もしくは，古いバージョンである場合は自動的にインストールされますが，インターネットにつながっている必要があります．

   Anaconda をご利用の場合は Using wheel files with conda もご確認ください．

4. 動作確認

   PySIMPLE がインストールされていれば以下でバージョンを確認することができます．：

```
$ python -m pysimple --version
pysimple 1.6.0
```

   続けて次のコマンドを実行してください．標準出力に大量の出力がされますが，下のような出力が最後に表示されていれば正しく動作しています．：

```
$ python -m pysimple.sample.sudoku
：
+-------+-------+-------+
| 5 3 4 | 6 7 8 | 9 1 2 |
| 6 7 2 | 1 9 5 | 3 4 8 |
| 1 9 8 | 3 4 2 | 5 6 7 |
+-------+-------+-------+
| 8 5 9 | 7 6 1 | 4 2 3 |
| 4 2 6 | 8 5 3 | 7 9 1 |
| 7 1 3 | 9 2 4 | 8 5 6 |
+-------+-------+-------+
| 9 6 1 | 5 3 7 | 2 8 4 |
| 2 8 7 | 4 1 9 | 6 3 5 |
| 3 4 5 | 2 8 6 | 1 7 9 |
+-------+-------+-------+
```

1.4 アップデート

インストール時と同じです．ダウングレードを行う場合は一旦アンインストールが必要です．

1.5 アンインストール

アンインストールはインストールと同じく pip コマンドで行います．以下のコマンドを実行してください．：

```
$ pip uninstall pysimple
Found existing installation: pysimple 1.6.0
Uninstalling pysimple-1.6.0:
  Would remove:
    c:\users\uname\appdata\local\programs\python\python3x\lib\site-packages\pysimple-
↪1.6.0.dist-info\*
    c:\users\uname\appdata\local\programs\python\python3x\lib\site-packages\pysimple\
↪kernel_wrapper.cp3x-win_amd64.pyd
    c:\users\uname\appdata\local\programs\python\python3x\lib\site-packages\pysimple\
↪sample\*
    c:\users\uname\appdata\local\programs\python\python3x\lib\site-packages\pysimple\
↪version.py
    c:\users\uname\appdata\local\programs\python\python3x\license
Proceed (Y/n)?
```

「Proceed(Y/n)?」と表示されましたら，アンインストールを続ける場合は「Y」を入力しリターンキーを押してください．アンインストールが成功すると下記のように「Successfully uninstalled pysimple-1.6.0」と表示されます．：

```
    c:\users\uname\appdata\local\programs\python\python3x\license
Proceed (Y/n)? Y
Successfully uninstalled pysimple-1.6.0
```

第2章　チュートリアル

2.1 はじめに

Nuorium Optimizer は数理計画問題を解くための汎用ソルバであり，PySIMPLE は数理計画問題を記述するモデリング言語です．

本節は Nuorium Optimizer/PySIMPLE の基本的な機能に関するチュートリアルです．本節を一読していただければ，Nuorium Optimizer/PySIMPLE の基本的な利用方法がご理解いただけると思います．

2.1.1 数理計画問題とは

数理計画問題とは，「与えられた条件の下で，望ましさの尺度を表す何らかの関数の最小値（最大値）を求め，さらにその最小値（最大値）を与える不特定要素の値を決定する」という問題です．

上記における，「与えられた条件」は制約条件，「望ましさの尺度を表す関数」は目的関数，「不特定要素」は変数と一般に呼ばれています．この用語を用いて書き直すと，数理計画問題とは，「制約条件を満たす範囲における目的関数の最小値（最大値）及びその最小値（最大値）を与える変数を求める問題」といえます．

例えば，$x \geq 0$ において $3x+2$ の最小値を求める問題は，数理計画問題です．この場合，制約条件は $x \geq 0$，目的関数は $3x+2$，変数は x となります．

この問題は数理計画の世界では次のように書かれます：

- 目的関数： $3x+2 \rightarrow$ 最小化
- 制約条件： $x \geq 0$

考える間もなく，上記の数理計画問題の最もよい目的関数値は 2（$x=0$ のとき）となります．このときの変数の値を最適解と呼びます．

最適解を求めることを「数理計画問題を解く」あるいは「最適化する」といいます．

2.1.2 数理計画問題を解く

`sample.py` というファイルに以下を記述してみましょう．：

```
from pysimple import Problem, Variable

problem = Problem(type=min)
x = Variable()
```

(次のページに続く)

(前のページからの続き)

```
problem += 3*x + 2
problem += x >= 0
problem.solve()
```

この状態で sample.py のあるフォルダに行き，コマンドプロンプトから python sample.py と入力してみましょう．：

```
$ python sample.py

[About Nuorium Optimizer]
Nuorium Optimizer 27.1.0 build:7491abe3
         <with META-HEURISTICS engine "wcsp"/"rcpsp">
         <with Netlib BLAS>
, Copyright (C) 1991 NTT DATA Mathematical Systems Inc.

[Problem and Algorithm]
PROBLEM_NAME                                        Problem
NUMBER_OF_VARIABLES                                       1
NUMBER_OF_FUNCTIONS                                       2
PROBLEM_TYPE                                    MINIMIZATION
METHOD                                          HIGHER_ORDER

[Progress]
<preprocess begin>.........<preprocess end>
<iteration begin>
    res=3.0e-03 .... 2.1e-10
<iteration end>

[Result]
STATUS                                              OPTIMAL
VALUE_OF_OBJECTIVE                                        2
ITERATION_COUNT                                           5
FUNC_EVAL_COUNT                                           8
FACTORIZATION_COUNT                                       6
RESIDUAL                                    2.077775694e-10
ELAPSED_TIME(sec.)                                     0.01
```

この一連の操作で，あなたは Nuorium Optimizer を使って次の数理計画問題を解いたことになります．ここで python は PySIMPLE をインストールしたときの Python 環境です．以降，python コマンドは PySIMPLE をインストールしたときの Python 環境を指します．

インタプリタから実行することもできます．python -i と入力してインタプリタを起動させてみましょう．：

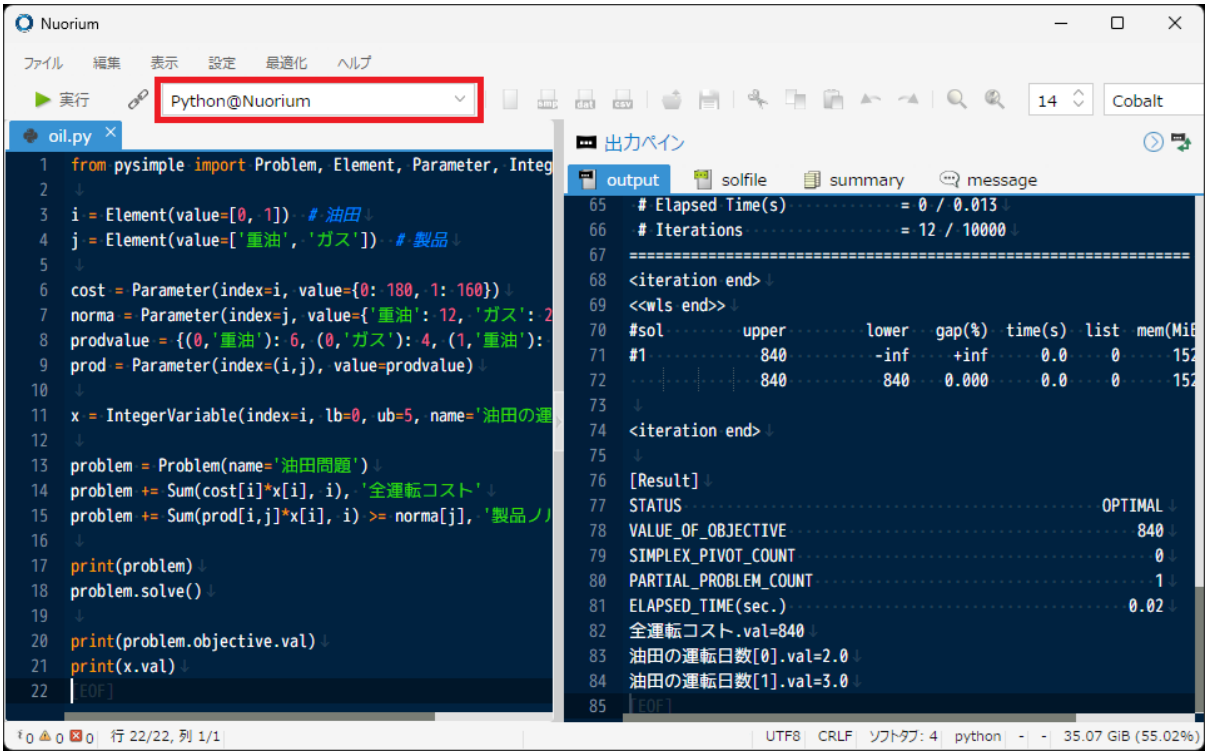
```
$ python -i
Python 3.13.2 (tags/v3.13.2:4f8bb39, Feb  4 2025, 15:23:48) [MSC v.1942 64 bit␣
↪(AMD64)] on win32
```

(次のページに続く)

(前のページからの続き)

```
Type "help", "copyright", "credits" or "license" for more information.
>>> from pysimple import Problem, Variable
pysimple 1.6.0 (2025-02-17 16:12:35 +0900 c93cf43)
Copyright (C) 2019 NTT DATA Mathematical Systems Inc. All Rights Reserved.
>>> problem = Problem(type=min)
>>> x = Variable()
>>> problem += 3*x + 2
>>> problem += x >= 0
>>> problem.solve()
# 以下同様の出力
```

Windows 環境では Nuorium 統合環境 からも実行できます．インストール時に PySIMPLE が予めインストールされた Python が同梱されており，これは「Python@Nuorium」を選択した状態で実行ボタン (F5) を押すことで実行できます．



ユーザが指定する Python を用いる場合には，以下の方法で実行できます．

python コマンドにパスが通っている場合

> 「デフォルト」を選択して実行ボタン (F5) を押してください．現在編集中の拡張子を見て，.py である場合にはパスが通っている Python を呼び出します．なお，このような拡張子ごとの「デフォルト」選択時の挙動は，ユーザフォルダ\AppData\Roaming\nuorium\editor\defaultrc.json から設定できます．

python コマンドにパスが通っていない場合

> 例えば python が「ユーザフォルダ\\Anaconda3」に存在する場合，ユーザフォルダ\AppData\Roaming\nuorium\editor\userrc.json を次のように変更して，新たに「Python」という実

行単位を追加します.

```
{
    "Python": {
        "command": "ユーザフォルダ\\Anaconda3\\python",
        "args": ["|fullname|"],
        "options": {
            "cwd": "|directory|"
        }
    }
}
```

ここで用いた実行単位名「Python」は例であり，他の区別可能な名称を用いて構いません.

以上に述べました詳細については，Nuorium 上で [メニューバー] → [ヘルプ] → [Nuorium リファレンスマニュアル (PDF)](F1) の 3.6 節「JSON ファイルの設定方法」をご確認ください.

Nuorium 上でターミナルを起動して，コマンドを実行することもできます．Ctrl-P を押すか，[メニューバー] → [設定] → [シェル設定] で開くシェル設定画面からターミナルを起動します．起動後，Nuorium 上にプロンプト画面が表示されますので，PySIMPLE をインストールした Python を呼び出します.

2.2 数理計画問題を記述する

本節では，具体的な例題を通して，モデリング言語 PySIMPLE の文法を紹介します.

Nuorium Optimizer C++SIMPLE マニュアルのチュートリアルに対応しておりますので C++SIMPLE による記述はそちらもご確認ください.

2.2.1 目的関数・変数・制約

次のような生産計画問題を考えます.

---

2 つの油田 X, Y が存在し，それぞれ一日あたり重油・ガスを次の量だけ生産する.

表 2.1: 生産量/日

	重油	ガス
X	6t	4t
Y	1t	6t

また，重油・ガスの週あたりの生産ノルマが，次のように定められている.

	ノルマ/週
重油	12t
ガス	24t

油田 X, Y の日あたりの運転コストは，次のとおりである．

	運転コスト/日
X	180
Y	160

油田 X, Y ともに，最大で週 5 日まで運転可能である．ノルマを満たしながら運転コストを最小化するためには，それぞれの油田を週あたり何日運転すれば良いだろうか？

この問題を定式化すると，以下のようになります．

変数	
x	油田 X の運転日数/週
y	油田 Y の運転日数/週

目的関数 (最小化)	
$180x + 160y$	運転コスト/週

制約条件	
$6x + y \geq 12$	重油ノルマ/週
$4x + 6y \geq 24$	ガスノルマ/週
$0 \leq x \leq 5$	油田 X の週あたりの運転日数制約
$0 \leq y \leq 5$	油田 Y の週あたりの運転日数制約

それでは，この問題を PySIMPLE で記述した例を見てみましょう．：

```
from pysimple import Problem, Variable

problem = Problem(name='油田問題 1')

# 油田 X, Y の運転日数/週（変数）
x = Variable(name='油田 X の運転日数')
y = Variable(name='油田 Y の運転日数')

# 運転コスト (目的関数)
problem += 180*x + 160*y, '全運転コスト'

# 製品ノルマ
problem += 6*x +   y >= 12, '重油ノルマ/週'
problem += 4*x + 6*y >= 24, 'ガスノルマ/週'

# 各油田の日数制約
```

(次のページに続く)

(前のページからの続き)

```
problem += 0 <= x, '油田 X の週あたりの運転日数制約 (下限)'
problem += x <= 5, '油田 X の週あたりの運転日数制約 (上限)'
problem += 0 <= y, '油田 Y の週あたりの運転日数制約 (下限)'
problem += y <= 5, '油田 Y の週あたりの運転日数制約 (上限)'

# 求解
print(problem)
problem.solve()

# 結果出力
print(x.val)
print(y.val)
print(problem.objective.val)
```

この PySIMPLE による記述を上から順に見ていきましょう．:

```
from pysimple import Problem, Variable
```

この部分はモデリングに必要なクラスや関数を使える状態にしています．PySIMPLE でモデリングを行う前には使用するオブジェクトを使える状態にしておく必要があります．インポートできるオブジェクト一覧やインポート方法はさまざまですが，ここでは一旦先に進むことにします．:

```
problem = Problem(name='油田問題 1')
```

この部分は問題の宣言です．name='..' には問題の名前を指定します．name='..' は省略可能ですが，出力などで使用されますので，なるべく記述した方が良いでしょう．

また，ここでは記述がありませんが，type=.. で目的関数の最小化・最大化を指定することができます．指定する場合は type=min や type=max のように記述します．省略した場合は最小化となります．:

```
# 油田 X, Y の運転日数/週（変数）
x = Variable(name='油田 X の運転日数')
y = Variable(name='油田 Y の運転日数')
```

この部分は変数（油田の運転日数）の宣言です．モデル中で使用する変数は，使用する前に宣言する必要があります．name='..' には変数の名前を指定しますが省略可能です．

から行の終わりまではコメントです．

```
# 運転コスト (目的関数)
problem += 180*x + 160*y, '全運転コスト'
```

この部分は目的関数（運転コスト）の内容を問題に設定しています．+= の左辺に問題を，右辺に目的関数とその名前を記述します．目的関数の名前は省略可能です．

* は積，+ は和を表す演算子です．PySIMPLE では四則演算や数学関数（Sum(), Exp(), ..）などを式の記述に用いることができます．

```
# 製品ノルマ
problem += 6*x +   y >= 12, '重油ノルマ/週'
problem += 4*x + 6*y >= 24, 'ガスノルマ/週'
```

この部分では制約式（生産ノルマ）を問題に設定しています．+= の左辺に問題を，右辺に制約式とその名前を記述します．制約式の名前は省略可能です．

関係演算子 >= の左辺，右辺には，任意の式を記述できます．目的関数の内容定義の際と同様に，任意の式の中に演算子や数学関数を記述できます．左辺と右辺の関係を表す関係演算子には，>=, <=, == を指定できます．:

```
# 各油田の日数制約
problem += 0 <= x, '油田 X の週あたりの運転日数制約 (下限)'
problem += x <= 5, '油田 X の週あたりの運転日数制約 (上限)'
problem += 0 <= y, '油田 Y の週あたりの運転日数制約 (下限)'
problem += y <= 5, '油田 Y の週あたりの運転日数制約 (上限)'
```

この部分は制約式（運転日数の上下限）を設定しています．problem += 0 <= x <= 5 のように一度に記述することはできません．変数の上下限は宣言時に下限を lb, 上限を ub で記述することもできます．上記の制約式を記述する代わりに変数を宣言するときに次のように記述します．:

```
x = Variable(lb=0, ub=5, name='油田 X の運転日数')
y = Variable(lb=0, ub=5, name='油田 Y の運転日数')
```

以上で，問題の定義の記述は完了です．

次に，これまでに定義した問題の最適解を求め，結果を出力する部分を記述します．:

```
# 求解
print(problem)
problem.solve()
```

print(問題) は問題に設定された情報を出力します．問題.solve() は定義した問題について最適解の計算を行う関数です．問題.solve() は，必ずモデル記述の後に記述する必要があります．

```
# 結果出力
print(x.val)
print(y.val)
print(problem.objective.val)
```

この部分は，最適化計算結果の出力を指定しています．変数や式に .val を付けることでその現在値を取り出すことができます．問題の目的関数は 問題.objective で参照できます．最適化計算後の値を出力するためには，最適化計算 solve の後に記述する必要があります．

以上でこのモデルについての PySIMPLE の記述は終了です．

次にこのモデルを実行してみます（実行方法については **数理計画問題を解く** を参照してください）．すると，数理計画モデルを解く経過が，以下のように出力されます．

```
Problem(name=' 油田問題 1', type=min):
[constraints]
重油ノルマ/週:
6*油田 X の運転日数+油田 Y の運転日数>=12
ガスノルマ/週:
4*油田 X の運転日数+6*油田 Y の運転日数>=24

[objective]
全運転コスト:
180*油田 X の運転日数+160*油田 Y の運転日数



[About Nuorium Optimizer]
Nuorium Optimizer 27.1.0 build:7491abe3
         <with META-HEURISTICS engine "wcsp"/"rcpsp">
         <with Netlib BLAS>
, Copyright (C) 1991 NTT DATA Mathematical Systems Inc.

[Problem and Algorithm]
PROBLEM_NAME                                    油田問題 1
NUMBER_OF_VARIABLES                                    2
NUMBER_OF_FUNCTIONS                                    3
PROBLEM_TYPE                                  MINIMIZATION
METHOD                                        HIGHER_ORDER

[Progress]
<preprocess begin>.........<preprocess end>
<iteration begin>
    res=1.2e+02 .... 4.1e-06  2.1e-08
<iteration end>

[Result]
STATUS                                             OPTIMAL
VALUE_OF_OBJECTIVE                                     750
ITERATION_COUNT                                          6
FUNC_EVAL_COUNT                                          9
FACTORIZATION_COUNT                                      7
RESIDUAL                                  2.072696939e-08
ELAPSED_TIME(sec.)                                    0.02
油田 X の運転日数.val=1.500000000256067
油田 Y の運転日数.val=2.9999999999710107
全運転コスト.val=750.0000000414537
```

最後に結果出力に対応する結果が以下のように出力されます．：

```
油田 X の運転日数.val=1.500000000256067
油田 Y の運転日数.val=2.9999999999710107
全運転コスト.val=750.0000000414537
```

+= の左辺は指定した変数と目的関数の名前で，name='..' に記述したものが出力されます．右辺には変数と目的関数の値が出力されています．

また，このモデルは PySIMPLE のサンプルとして同梱されています．このサンプルを実行するには次のようにします．:

```
$ python -m pysimple.sample.tutorial oil1
```

2.2.2 定数

現在は，モデル中に油田運転コストの値を直接記述しています．これを変更し，外部から任意の値を与えてみましょう．まず，定式化を以下のように変更します．

目的関数	
$costX \cdot x + costY \cdot y$	運転コスト/週

定数	
$costX$	油田 X の運転コスト/日
$costY$	油田 Y の運転コスト/日

$costX$, $costY$ はそれぞれ油田 X, Y の運転コスト/日を表す定数です．PySIMPLE では，このような定数を使用した記述が可能です．

ここでは，定数を用いて，運転コストを以下のように変更します．:

```
from pysimple import Problem, Variable
```

↓

```
from pysimple import Problem, Variable, Parameter
```

```
problem += 180*x + 160*y, '全運転コスト'
```

↓

```
costX = Parameter(value=180, name='油田 X の運転コスト')
costY = Parameter(value=160, name='油田 Y の運転コスト')
problem += costX*x + costY*y, '全運転コスト'
```

まず，Parameter で定数を宣言します．モデル中で使用する定数は，使用する前に宣言する必要があります．value=.. には定数の値を指定します．name='..' には定数の名前を指定しますが省略可能です．

今回のモデルでは添字がないので Parameter を用いずに記述しても構いません．:

```
costX = 180
costY = 160
problem += costX*x + costY*y, '全運転コスト'
```

では，実行してみます（実行方法については **数理計画問題を解く** を参照してください）．

最適化経過の出力の後，次のような実行結果が得られます．:

```
油田 X の運転日数.val=1.500000000256067
油田 Y の運転日数.val=2.9999999999710107
全運転コスト.val=750.0000000414537
```

前回と同じ結果が得られています．Parameter を使用することで，データの変更のみで違う問題を解くことができます．

では，データを変更して実行してみましょう．以下のようにデータを変更します．:

```
costX = Parameter(value=100, name='油田 X の運転コスト')
costY = Parameter(value=170, name='油田 Y の運転コスト')
```

実行すると，以下の結果が得られます．:

```
油田 X の運転日数.val=4.99999999387777
油田 Y の運転日数.val=0.6666666712283269
全運転コスト.val=613.3333334965926
```

また，このモデルは PySIMPLE のサンプルとして同梱されています．このサンプルを実行するには次のようにします．:

```
$ python -m pysimple.sample.tutorial oil2
```

2.2.3 集合・添字

油田運転日数を一般的に記述することを考えてみましょう．まず油田運転日数 x, y をそれぞれ x_0, x_1 と変更し，定式化を次のように変更します．

集合，添字	
$i \in OilField = \{0, 1\}$	油田集合，添字

変数	
$x_i, \quad i \in OilField$	油田 i の運転日数/週

定数	
$costX$	油田 0 の運転コスト/日
$costY$	油田 1 の運転コスト/日

目的関数	
$costX \cdot x_0 + costY \cdot x_1$	運転コスト/週

制約条件	
$6x_0 + x_1 \geq 12$	重油ノルマ/週
$4x_0 + 6x_1 \geq 24$	ガスノルマ/週
$0 \leq x_i \leq 5, \quad \forall i \in OilField$	油田 i の週あたりの運転日数制約

対応する PySIMPLE の記述は，次のようになります．:

```
from pysimple import Problem, Element, Parameter, Variable

problem = Problem(name='油田問題 3')

# 油田を表す添字
i = Element(value=[0, 1], name='油田')

# 油田 i の運転コスト/日
costX = Parameter(value=180, name='油田 X の運転コスト')
costY = Parameter(value=160, name='油田 Y の運転コスト')

# 油田 i の運転日数（変数）
x = Variable(lb=0, ub=5, index=i, name='油田の運転日数')

# 運転コスト/週（目的関数）
problem += costX*x[0] + costY*x[1], '全運転コスト'

# 製品ノルマ
problem += 6*x[0] +   x[1] >= 12, '重油ノルマ/週'
problem += 4*x[0] + 6*x[1] >= 24, 'ガスノルマ/週'

# 求解
print(problem)
```

(次のページに続く)

(前のページからの続き)

```
problem.solve()

# 結果出力
print(x.val)
print(problem.objective.val)
```

定式化と同様，日数制約を一度に書き表しています．

それでは，PySIMPLE の記述の変更・追加点について，上から順に見ていきます．:

```
# 油田を表す添字
i = Element(value=[0, 1], name='油田')
```

ここでは添字（油田の添字）を宣言しています．value=.. には添字がとる要素の列を指定します．ここでは先の定式化の添字範囲が {0, 1} なので，0, 1 を添字の範囲とします．name='..' には添字の名前を指定しますが省略可能です．

C++SIMPLE のように集合を作成した後に，添字を作成することもできます．:

```
# 油田集合
OilField = Set(value=[0, 1], name='油田集合')
i = Element(set=OilField)
```

Set を用いる場合には Element と同様にインポートが必要になります．:

```
x = Variable(lb=0, ub=5, index=i, name='油田の運転日数')
```

ここでは油田の運転日数を，添字付き変数として宣言しています．index=.. で添字を指定します．

```
# 運転コスト/週（目的関数）
problem += costX*x[0] + costY*x[1], '全運転コスト'
```

ここでは運転コストの内容定義をしています．添字付けは，x[添字] と記述します．:

```
# 製品ノルマ
problem += 6*x[0] +   x[1] >= 12, '重油ノルマ/週'
problem += 4*x[0] + 6*x[1] >= 24, 'ガスノルマ/週'
```

ここでは製品ノルマの制約を記述しています．以前に x, y と書いた変数部分を x[0], x[1] と置き換えただけです．

```
# 結果出力
print(x.val)
```

結果出力も上記日数制約と同様に，添字に i と指定することで，全ての $i \in OilField$ について x[i] の値が出力されます．ここでは print(x.val) としていますが，print(x[i].val) でも同じ効果があります．

次に実行してみます（実行方法については **数理計画問題を解く** を参照してください）．最適化経過が出力されたあと，print(x) に対応した，以下の出力が得られます．

```
油田の運転日数[0].val=1.5000000000256067
油田の運転日数[1].val=2.9999999999710107
```

変数名が添字つきで出力されているのが確認できます.

ここまでの記述の変更で，添字 i を導入し，各油田の運転日数を x[i] と簡略化することができました．次に，油田運転コスト costX, costY も添字 i を用いて簡略化してみます．運転コストを添字付けし，以下のように表すことにします.

定数
$costX_i, i \in OilField$　油田 i の運転コスト/日

$costX_0$, $costX_1$ はそれぞれ以前の $costX$, $costY$ に対応する定数です．PySIMPLE でも同様に定数の添字付けを用いて，以下のように修正します．:

```
# 油田 i の運転コスト/日
costX = Parameter(value=180, name='油田Xの運転コスト')
costY = Parameter(value=160, name='油田Yの運転コスト')

# 運転コスト/週（目的関数）
problem += costX*x[0] + costY*x[1], '全運転コスト'
```

↓

```
# 油田 i の運転コスト/日
costX = Parameter(index=i, value={0: 180, 1: 160}, name='油田運転コスト')

# 運転コスト/週（目的関数）
problem += costX[0]*x[0] + costX[1]*x[1], '全運転コスト'
```

定数の添字付けは，変数の添字付けと同様に `index=i` と指定します．定数の値を要素ごとに指定する場合は Python の辞書で指定します.

では，実行してみましょう（実行方法については **数理計画問題を解く** を参照してください）．最適化経過が出力された後，以下のように以前と同様の結果が得られます．:

```
油田の運転日数[0].val=1.5000000000256067
油田の運転日数[1].val=2.9999999999710107
```

次に，重油とガスの製品についても一般に記述してみましょう．定式化において製品集合を導入して製品ノルマを以下のように記述します.

集合	
$Product = \{$ 重油, ガス $\}$	製品集合

定数	
$norma_j, \quad j \in Product$	製品 j のノルマ/週

PySIMPLE の記述においても同様に定数の添字付けを用いて表現し，ノルマに関する制約式を以下のように変更します．：

```
# 製品ノルマ
problem += 6*x[0] +   x[1] >= 12, '重油ノルマ/週'
problem += 4*x[0] + 6*x[1] >= 24, 'ガスノルマ/週'
```

↓

```
# 製品を表す添字
j = Element(value=['重油', 'ガス'], name='製品')

# 製品 j のノルマ/週
norma = Parameter(index=j, value={'重油': 12, 'ガス': 24}, name='製品ノルマ')

# 製品ノルマ
problem += 6*x[0] +   x[1] >= norma['重油'], '重油ノルマ/週'
problem += 4*x[0] + 6*x[1] >= norma['ガス'], 'ガスノルマ/週'
```

新たに製品を表す添字 j の宣言を追加し，製品ノルマを添字 j 付きの定数にします．上記のように文字列を添字に使用する場合は，文字列を '..' の中に記述する必要があります．

ここまでの変更をまとめて，集合，変数，定数，制約条件，目的関数を分類し整理すると，定式化と PySIMPLE の記述は次のようになります．

集合，添字

$i \in OilField = \{0, 1\}$	油田集合，添字
$j \in Product = \{$ 重油, ガス $\}$	製品集合，添字

定数

$costX_i, \quad i \in OilField$	油田 i の運転コスト/日
$norma_j, \quad j \in Product$	製品 j のノルマ/週

変数

$x_i, \quad i \in OilField$	油田 i の運転日数/週

目的関数（最小化）

$costX_0 \cdot x_0 + costX_1 \cdot x_1$	運転コスト/週

制約条件

$6x_0 + x_1 \geq norma_{重油}$	重油ノルマ/週
$4x_0 + 6x_1 \geq norma_{ガス}$	ガスノルマ/週
$0 \leq x_i \leq 5, \quad \forall i \in OilField$	油田 i の週あたりの運転日数制約

```
from pysimple import Problem, Element, Parameter, Variable

problem = Problem(name=' 油田問題 3')

# 油田を表す添字
i = Element(value=[0, 1], name=' 油田')
## 油田集合
#OilField = Set(value=[0, 1], name='油田集合')
#i = Element(set=OilField)

# 製品を表す添字
j = Element(value=[' 重油', ' ガス'], name=' 製品')
## 製品集合
#Product = Set(value=['重油', 'ガス'], name='製品集合')
#j = Element(set=Product)

# 油田 i の運転コスト/日
costX = Parameter(index=i, value={0: 180, 1: 160}, name=' 油田運転コスト')

# 製品 j のノルマ/週
norma = Parameter(index=j, value={' 重油': 12, ' ガス': 24}, name=' 製品ノルマ')

# 油田 i の運転日数（変数）
x = Variable(lb=0, ub=5, index=i, name=' 油田の運転日数')
```

(次のページに続く)

(前のページからの続き)

```
print(costX)
print(norma)

# 運転コスト/週（目的関数）
problem += costX[0]*x[0] + costX[1]*x[1], '全運転コスト'

# 製品ノルマ
problem += 6*x[0] +   x[1] >= norma['重油'], '重油ノルマ/週'
problem += 4*x[0] + 6*x[1] >= norma['ガス'], 'ガスノルマ/週'

# 求解
print(problem)
problem.solve()

# 結果出力
print(x.val)
print(problem.objective.val)
```

このモデルは PySIMPLE のサンプルとして同梱されています．このサンプルを実行するには次のようにします．:

```
$ python -m pysimple.sample.tutorial oil3
```

2.2.4 集約・複数の添字

コスト定義式:

```
# 運転コスト/週（目的関数）
problem += costX[0]*x[0] + costX[1]*x[1], '全運転コスト'
```

は，すべての油田について運転コストの和をとるという意味なので，これを一般的に記述すると，以下のようになります．

$$\sum_i costX_i \cdot x_i$$

対応する PySIMPLE の記述は，以下のようになります．:

```
Sum(costX[i]*x[i], i)
```

Sum() は $\sum$ に対応する関数で，:

```
Sum(和をとる式，添字)
```

の書式を持ちます．

次にノルマ制約についても，Sum() を適用したいと考えますが，旧記述では，：

```
# 製品ノルマ
problem += 6*x[0] +   x[1] >= norma[' 重油'], ' 重油ノルマ/週'
problem += 4*x[0] + 6*x[1] >= norma[' ガス'], ' ガスノルマ/週'
```

と各油田の生産量が直接数値で記述されているので，一般化できません．そこで，定式化において定数 $prodX_{i,j}$ を導入し，制約式を次のように記述します．

制約条件	
$\sum_{i \in OilField} prodX_{i,j} \cdot x_i \geq norma_j, \quad \forall j \in Product$	製品 j のノルマ/週の制約式

定数	
$prodX_{i,j}, \quad i \in OilField, j \in Product$	油田 i の製品 j 生産量/日
$norma_j, \quad j \in Product$	製品 j のノルマ/週

対応する PySIMPLE の記述は，以下のようになります．：

```
# 油田 i の製品 j 生産量/日
prodXvalue = {(0,' 重油'): 6, (0,' ガス'): 4,
              (1,' 重油'): 1, (1,' ガス'): 6}
prodX = Parameter(index=(i,j), value=prodXvalue, name=' 油田の生産量')

# 製品ノルマ
problem += Sum(prodX[i,j]*x[i], i) >= norma[j], ' 製品ノルマ'
```

複数の添字に依存する定数を宣言する際には，`index=(i,j,..)` と指定します．複数の添字に対する値は辞書のキーをタプルにします．

`Sum()` は指定した添字 i のみの和をとります．i, j について和をとる場合は，`Sum(任意の式，(i,j))` と記述します．

実行結果は以前と同様になります．

ここまでの変更をまとめて，添字，変数，定数，制約条件，目的関数を分類し整理すると，定式化と PySIMPLE の記述は次のようになります．

添字	
$i \in OilField = \{0, 1\}$	油田を表す添字
$j \in Product = \{$ 重油, ガス $\}$	製品を表す添字

定数	
$costX_i, \quad i \in OilField$	油田 i の運転コスト/日
$norma_j, \quad j \in Product$	製品 j のノルマ/週
$prodX_{i,j}, \quad i \in OilField, j \in Product$	油田 i の製品 j 生産量/日

変数	
$x_i, \quad i \in OilField$	油田 i の運転日数/週

目的関数（最小化）	
$\sum_{i \in OilField} costX_i \cdot x_i$	運転コスト/週

制約条件	
$\sum_{i \in OilField} prodX_{i,j} \cdot x_i \geq norma_j$	製品 j のノルマ/週の制約式
$0 \leq x_i \leq 5, \forall i \in OilField$	油田 i の週あたりの運転日数制約

```
from pysimple import Problem, Element, Parameter, Variable, Sum

problem = Problem(name='油田問題 4')

# 油田を表す添字
i = Element(value=[0, 1], name='油田')

# 製品を表す添字
j = Element(value=['重油', 'ガス'], name='製品')

# 油田 i の運転コスト/日
costX = Parameter(index=i, value={0: 180, 1: 160}, name='油田運転コスト')

# 製品 j のノルマ/週
norma = Parameter(index=j, value={'重油': 12, 'ガス': 24}, name='製品ノルマ')

# 油田 i の製品 j 生産量/日
prodXvalue = {(0,'重油'): 6, (0,'ガス'): 4,
              (1,'重油'): 1, (1,'ガス'): 6}
prodX = Parameter(index=(i,j), value=prodXvalue, name='油田の生産量')

# 油田 i の運転日数（変数）
x = Variable(lb=0, ub=5, index=i, name='油田の運転日数')
```

(次のページに続く)

(前のページからの続き)

```
print(costX)
print(norma)
print(prodX)

# 運転コスト/週（目的関数）
problem += Sum(costX[i]*x[i], i), '全運転コスト'

# 製品ノルマ
problem += Sum(prodX[i,j]*x[i], i) >= norma[j], '製品ノルマ'

# 求解
print(problem)
problem.solve()

# 結果出力
print(x.val)
print(problem.objective.val)
```

このモデルは PySIMPLE のサンプルとして同梱されています．このサンプルを実行するには次のようにします．:

```
$ python -m pysimple.sample.tutorial oil4
```

2.2.5 式

ここでは，これまでの結果出力（油田運転日数/週, 全運転コスト）に加えて，各製品の生産量/週も出力してみます．

$$prod_j = \sum_{i \in OilField} prodX_{i,j} \cdot x_i, \forall j \in Product \quad \text{製品 } j \text{ の生産量/週}$$

この式に対応する PySIMPLE の記述は，以下のようになります．:

```
prod = Sum(prodX[i,j]*x[i], i)
prod.name = '製品の生産量'
```

PySIMPLE では式は宣言なしに扱うことができます．name 属性は自動で与えられますが，上記のようにして変更することもできます．また，index も自動で与えられます．上記の場合，i で和をとっているので prod の index は j となります．

次に生産ノルマの記述を見てみます．:

```
problem += Sum(prodX[i,j]*x[i], i) >= norma[j], '製品ノルマ'
```

左辺は先ほど定義した prod と全く同じ内容ですので，以下のように左辺を prod に置き換えることができます．

```
problem += prod >= norma[j], '製品ノルマ'
```

prod は単なる置き換えなので，上記のように添字を用いなくても構いませんが，添字を明記することもできます．:

```
problem += prod[j] >= norma[j], '製品ノルマ'
```

次に結果出力部分に以下のように prod を追加します．:

```
print(prod.val)
```

同様に添字を明記する場合は print(prod[j].val) と記述してください．

これで，製品の生産量/週が出力されるようになりました．生産量の出力結果は，以下のようになります．:

```
製品の生産量 [重油].val=12.000000001507413
製品の生産量 [ガス].val=24.000000000850335
```

このモデルは PySIMPLE のサンプルとして同梱されています．このサンプルを実行するには次のようにします．:

```
$ python -m pysimple.sample.tutorial oil5
```

2.2.6 整数変数

ここまでは，運転日数を連続変数とみなして解いてきました．しかし実際には油田は 1 日単位でしか運転できません．そこで，運転日数を 1 日単位の整数変数とした，整数計画問題を解くことを考えます．そのために，変数（運転日数）の宣言を以下のように変更します．:

```
from pysimple import Problem, Element, Parameter, Variable, Sum
```

↓

```
from pysimple import Problem, Element, Parameter, IntegerVariable, Sum
```

```
# 油田 i の運転日数（変数）
x = Variable(lb=0, ub=5, index=i, name='油田の運転日数')
```

↓

```
# 油田 i の運転日数（変数）
x = IntegerVariable(lb=0, ub=5, index=i, name='油田の運転日数')
```

IntegerVariable で整数変数を宣言します．整数変数として宣言された変数は，値として整数のみを取ります．以上で変更完了です．

実行すると，以下の結果が得られます．:

```
油田の運転日数 [0].val=2.0
油田の運転日数 [1].val=3.0
製品の生産量 [重油].val=15
製品の生産量 [ガス].val=26
全運転コスト.val=840
```

運転日数が整数になっているのが確認できます．このように変数を IntegerVariable で宣言するだけで，整数計画問題を記述することができます．

このモデルは PySIMPLE のサンプルとして同梱されています．このサンプルを実行するには次のようにします．:

```
$ python -m pysimple.sample.tutorial oil6
```

2.2.7 結果出力関数

ここまでは，結果の出力には Python の組み込み関数 `print()` を使用してきましたが，PySIMPLE は他にも書式指定出力関数 `Printf()` があります．

```
from pysimple import Problem, Element, Parameter, IntegerVariable, Sum
```

↓

```
from pysimple import Problem, Element, Parameter, IntegerVariable, Sum, Printf
```

```
# 結果出力
print(x.val)
```

↓

```
# 結果出力
Printf('油田 {} の最適運転日数 = {:.0f}', i, x[i].val)
```

対応する実行結果出力は以下のようになります．:

```
油田 0 の最適運転日数 = 2
油田 1 の最適運転日数 = 3
```

関数 `Printf()` の書式指定は，:

```
Printf(出力書式指定，出力対象 1，出力対象 2，...)
```

となります．

出力対象には，添字，変数，式，定数，目的関数，など任意のものを任意の個数だけ指定できます．出力書式指定の指定方法は，*Printf* を確認してください．

このモデルは PySIMPLE のサンプルとして同梱されています．このサンプルを実行するには次のようにします．:

```
$ python -m pysimple.sample.tutorial oil7
```

2.2.8 デバッグ出力関数

数理計画モデルが複雑になるほど，些細な記述ミスでも発見が困難になっていきます．そのようなミスを修正するための支援関数としても print() 関数や Printf() 関数は有効です．

以下のように，print(problem) を最適化計算 problem.solve() の直前に挿入してみます．

```
print(problem)
problem.solve()
```

上記の位置に記述すれば，最適化計算を行うモデルの内容が出力できます．これを実行すると，print() に対応した出力が以下のように得られます．

```
Problem(name='油田問題 7', type=min):
[constraints]
製品ノルマ:
6*油田の運転日数 [0]+油田の運転日数 [1]>=12
4*油田の運転日数 [0]+6*油田の運転日数 [1]>=24

[objective]
全運転コスト:
180*油田の運転日数 [0]+160*油田の運転日数 [1]
```

3〜5 行目は次のノルマ制約式に対応しています．:

```
problem += prod >= norma[j], '製品ノルマ'
```

7 行目以降は次のコスト定義式に対応しています．:

```
problem += Sum(costX[i]*x[i], i), '全運転コスト'
```

以下のように特定の制約式や目的関数のみを表示することもできます．:

```
print(problem['製品ノルマ'])
print(problem.objective)
```

このように，print() を使用することによって，定数値，添字等を実際の値に置き換えた後の目的関数・制約式を確認することができます．この機能を利用すれば，意図しない記述ミスを簡単に発見することができ，効率の良いモデル記述が可能になります．

第3章　ユーザガイド

3.1 PySIMPLE の基本事項

3.1.1 モデリング言語 PySIMPLE とは

PySIMPLE はシステムの記述をなるべく人間に馴染みのある数学的な記述方法で簡単に行ない，実際のシミュレータやソルバなどが認識できるような表現に翻訳して所要の解析を行うことを目的とします．

PySIMPLE はプログラミング言語 Python を用いて実装されています．PySIMPLE を用いるに際して Python の知識を特別必要とすることはありませんが，一部 Python を理解していないと利用が難しい機能もあります．

3.1.2 PySIMPLE の構成

最適化パッケージソフト Nuorium Optimizer のインターフェースとして PySIMPLE をご利用いただく場合，以下の構成となります．

- PySIMPLE（数理計画問題を記述するためのモデリング言語）
- Nuorium Optimizer（数理計画問題を解くための求解ソルバ）

本マニュアルで Nuorium Optimizer と呼ぶ場合，上記二つをまとめたソフト全体を指すケースと，後者の求解ソルバのみを指すケースがありますのでご注意ください．

3.1.3 PySIMPLE の利用法

PySIMPLE を用いて Nuorium Optimizer をご利用いただく場合，コマンドライン等の Python を起動させる方法にて実行させます．コマンドラインによる実行方法は **数理計画問題を解く** をご覧ください．

3.1.4 PySIMPLE の処理の流れ

PySIMPLE **の利用法** で取り上げたコマンドラインでの処理の流れを説明します．

1. PySIMPLE で数理計画問題（モデル）を記述
2. モデルを Python のスクリプトとして実行

数理計画問題（モデル）の記述はテキストエディタをご利用ください．

また，ファイル名には Windows のファイル名として使える文字が使用できますが，通常は半角英数字および半角アンダースコア（_）程度にしておきましょう．

3.1.5 ファイルの文字コード指定について

Python3 では ソースコードのデフォルトエンコーディングは UTF-8 なので，ソースコード中に Unicode 文字をそのまま含めることができます．

ソースコードの文字コードが UTF-8 でない場合，Python はソースコードの解釈に失敗することがあります．例えば，ソースコードの文字コードが shift_jis である場合，次のようなエラーが出ます．：

```
SyntaxError: Non-UTF-8 code starting with '…' in file …
```

これはソースコードを UTF-8 として解釈しようとしているためです．このような場合，次のような特殊な形式のコメントをソースコードの 1 行目もしくは 2 行目に配置することで，UTF-8 ではないエンコーディングを使うことができます．：

```
# -*- coding: shift_jis -*-
```

詳細は Python のソースコードの文字コードをご確認ください．

3.2 Python の基本事項

PySIMPLE を利用するに当たって，基本となる Python の文法を簡単に説明します．Python の詳細な文法については 公式マニュアル をご確認ください．

3.2.1 数値，文字列

整数型の int，浮動小数点型の float，文字列型の str などが存在します．文字列における「'」と「"」は同じです．：

```
>>> 2 + 3
5
>>> 'ほげ'
'ほげ'
>>> "ほげ"
'ほげ'
```

半角空白文字（半角スペース）と改行はモデル中では任意に用いる事ができます．全角空白文字（全角スペース）を用いる事はできません．

「#」から行末まではコメントになります．：

```
>>> 3.14  # 円周率
3.14
```

「変数名= オブジェクト」でオブジェクトを保持する変数を作成します．一度作成した変数に再度，異なるオブジェクトを代入することもできます．：

```
>>> x = 3.14
>>> x
3.14
>>> x = 'pi'
>>> x
'pi'
```

3.2.2 データ構造

リスト

リストは様々なオブジェクト含むことができるデータ構造です．[と] で囲んで，で要素を並べます．：

```
>>> lst = [3, 'spam', 3.14, [0, 'ham']]
>>> lst
[3, 'spam', 3.14, [0, 'ham']]
```

リストに対して [index] で index 番目の要素を取り出したり，修正することができます．ただし，最初の要素を 0 番目と数えます．：

```
>>> lst[1]
'spam'
>>> lst[1] = 'egg'
>>> lst[1]
'egg'
```

タプル

タプルはリストと似ていますが，値を変えることができません．

```
>>> tpl = (3, 'spam', 3.14, [0, 'ham'])
>>> tpl[1]
'spam'
>>> tpl[1] = 'egg'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

タプルはリストと異なり () で囲む必要がない場合もあるのですが，慣れない内は括弧を付けるようにしておきましょう．また，リストとタプルは list 関数, tuple 関数によりそれぞれ相互変換できます．：

```
>>> tpl = (1, 2, 3)
>>> lst = list(tpl)
>>> lst
[1, 2, 3]
>>> tuple(list(tpl))
(1, 2, 3)
```

辞書

辞書は対応付けのためのデータ構造です．**キー** と **値** を：で挟み，で並べて { と } で囲みます．：

```
>>> dct = {'spam': 1, 'ham': 2}
>>> dct['ham']
2
```

この場合，キーは 'spam' と 'ham'，値は 1 と 2 です．次の例はもう少し複雑で，キーがタプルになっています．：

```
>>> dct = {(1, 2): 3, (1, 3): 4}
>>> dct[(1,3)]
4
>>> dct[1,3]
4
```

dict 関数を用いることでタプルの列から辞書を作ることもできます．：

```
>>> dct = dict([('X', 1), ('Y', 2), ('Z', 3)])
>>> dct
{'X': 1, 'Y': 2, 'Z': 3}
```

3.2.3 組み込み関数

最初から使うことのできる関数は他にもいろいろあります．ここでは PySIMPLE を記述する上で有用なものを幾つか紹介します．

print, help, dir

print 関数は何でも表示してくれます．インタプリタでは明示的に記述しなくても大丈夫ですが，ファイルに記述する場合は，表示させたい箇所に print 関数を挟みましょう．:

```
x = 2
x = x + 3
print(x)  # 5
```

help 関数はオブジェクトの使い方を教えてくれます．print 関数の使い方を見てみましょう．:

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

dir 関数は属性のリストを返します．リストの属性一覧を見てみましょう．:

```
>>> lst = [1, 2]
>>> dir(lst)
['__add__', .., 'append', ..]
```

append という属性を持っていることが分かります．この使い方を調べて実際に使ってみましょう．属性はオブジェクトに対して . で続けます．:

```
>>> help(lst.append)
Help on built-in function append:

append(...) method of builtins.list instance
    L.append(object) -> None -- append object to end

>>> lst.append(3)
>>> lst
[1, 2, 3]
```

help のとおり要素が追加されました．

range

range 関数は等差数列を作るための関数です．range(stop) で 0 から stop-1 までの等差数列を，range(start, stop) で start から stop-1 までの等差数列を作ります．：

```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(2, 5))
[2, 3, 4]
```

zip

zip 関数を使うと 2 つのリスト等から辞書をつくるときに便利です．：

```
>>> list(zip(['X', 'Y', 'Z'], [1, 2, 3]))
[('X', 1), ('Y', 2), ('Z', 3)]
>>> dict(zip(['X', 'Y', 'Z'], [1, 2, 3]))
{'X': 1, 'Y': 2, 'Z': 3}
```

zip 関数は引数それぞれの 1 番目，2 番目…を取り出した列を作ります．

3.3 数理計画モデルの構成要素

以下は PySIMPLE を用いて数理計画モデルを記述する際の構成要素の一覧です．ここでは全ての構成要素を列挙してはいませんが，大半の数理計画モデルは以下の構成要素の組合せで記述することができます．

構成要素名	PySIMPLE 内の名称	機能
問題	Problem	問題を表す
集合	Set	添字の動く範囲を表す
添字	Element	添字を表す
定数	Parameter	定数を表す
変数	Variable	変数を表す
式		頻出する数式に対して，簡単な別の表現を与える
制約式	Constraint	制約式を表す
ハード制約関数	HardConstraint	ハード制約を表す
セミハード制約関数	SemiHardConstraint	セミハード制約を表す
ソフト制約関数	SoftConstraint	ソフト制約を表す
整数変数	IntegerVariable	整数変数を表す
0-1 整数変数	BinaryVariable	0-1 整数変数を表す
範囲演算関数	Sum，Prod	$\sum$ や $\prod$ に相当する
選択関数	Selection	限定選択を表す
条件式		制約式や代入文を制限する
数学関数	Exp，Sin，Cos，Log ...	数学関数を表す

3.3.1 問題クラス Problem

問題は Problem というクラスで表現されます．:

```
problem = Problem(type=min)
```

問題を定義する際には type キーワードに min あるいは max を指定することができます．これはそれぞれ扱う数理計画問題が最小化問題，最大化問題であることを意味します．この type= を省略した場合，自動的に最小化問題であると見做されます．

問題に対し，+= 式 を行うと目的関数と見做されます．:

```
x = Variable()
problem = Problem(type=min)
problem += 2*x
```

カンマで区切って文字列を与えることにより目的関数に名前をつけることもできます．:

```
problem += 2*x, '目的関数'
```

問題の目的関数は objective 属性でアクセスすることができます．:

```
print(problem.objective)
```

問題に対し，+= 制約式 を行うと制約式の追加と見做されます．:

```
problem += 2*x >= 0
```

カンマで区切って文字列を与えることにより制約式に名前をつけることもできます．:

```
problem += 2*x >= 0, '制約式 1'
```

問題の制約式は 問題 [制約式名] でアクセスすることができます．:

```
print(problem['制約式 1'])
```

目的関数に添字をつける事はできません．例えば，以下の記述は誤りです．:

```
i = Element(value=[1,2])
x = Variable(index=i)
problem = Problem(type=min)
problem += 2*x[i]
```

目的関数を複数設定することはできません．例えば，以下の記述は誤りです．:

```
x = Variable()
problem = Problem(type=min)
problem += 2*x
problem += 3*x + 1
```

solve 関数

求解（最適解計算の実行）を行うには solve メソッドを呼び出します．：

```
problem.solve()
```

solve 関数の前に変数や式の値を出力させると，求解前の初期状態の情報が記述されます．例えば，次のモデルに対する出力は以下のようになります．：

```
p = Problem(type=min)
i = Element(value=[1, 2, 3])
x = Variable(index=i)
p += Sum(2*x[i])
p += x[i] >= 5
x[i] = 10
print(x.val)  # 10
p.solve(silent=True)
print(x.val)  # 5
```

出力:

```
x[1].val=10
x[2].val=10
x[3].val=10
x[1].val=5.0000000020802062
x[2].val=5.0000000020802062
x[3].val=5.0000000020802062
```

solve 関数は，デフォルトでは標準出力に求解情報を表示し，解ファイル（モデル名.sol）を作成しません．標準出力による求解情報の表示を抑制するには，問題の宣言時か求解時に silent キーワードを記述します．：

```
p = Problem(silent=True)
```

```
p = Problem()
p.solve(silent=True)
```

求解時の設定は問題宣言時の設定に優先され，また一時的です．

解ファイルを出力するには solfile キーワードを記述します．解ファイル，及び実行不可能性要因検出機能については 実行不可能性要因検出機能 をご確認ください．：

```
p = Problem(solfile=True)
```

```
p = Problem()
p.solve(solfile=True)
```

その他の求解時の制御については 求解オプション や コールバック関数 をご確認ください．

最適化計算結果 result

最適化計算の結果は result メンバが保持しています．一覧は *pysimple.problem.Result* を確認してください．以下は最適化計算結果を出力する記述例です．:

```
x = Variable()
y = Variable()
p = Problem()
p += 2*x + 3*y
p += x + 2*y == 15
p += x >= 0
p += y >= 0
p.solve(silent=True)
print(p.result)
Printf(' 関数の数          {}', p.result.nfunc)
Printf(' 内点法の反復回数 {}', p.result.iters)
Printf(' 関数評価回数      {}', p.result.fevals)
Printf(' 目的関数値        {}', p.result.optValue)
Printf(' 収束判定条件      {}', p.result.tolerance)
Printf(' 最適性条件残差    {}', p.result.residual)
Printf(' 所要計算時間      {}', p.result.elapseTime)
Printf(' 終了時ステータス {}', p.result.errorCode)
```

出力:

```
errorMessage          : ''
method                : 'higher'
optValue              : 22.50000000376254
errorCode             : 0
nvars                 : 2
nfunc                 : 4
iters                 : 6
elapseTime            : 0.001999974250793457
peakPhysicalMemoryUsed: 44
peakVirtualMemoryUsed : 1518
fevals                : 9
factCount             : 7
tolerance             : 1e-08
residual              : 3.762538162000488e-09
infeasibility         : 3.552713678800501e-15
consInfeasibility     : 0.0
varInfeasibility      : 0.0
hardPenalty           : 0.0
semiHardPenalty       : 0.0
softPenalty           : 0.0
IIS                   : no IIS
```

(次のページに続く)

(前のページからの続き)

```
関数の数          4
内点法の反復回数  6
関数評価回数      9
目的関数値        22.500000000376254
収束判定条件      1e-08
最適性条件残差    3.762538162000488e-09
所要計算時間      0.001999974250793457
終了時ステータス  0
```

3.3.2 集合クラス Set

集合は Set というクラスで表現されます.

集合は value キーワードを用いて宣言します．PySIMPLE の集合はすべて順序集合です．また，集合の要素を宣言後に変更することはできません.

以下の例では，自然数 1, 2, 3 を要素とする集合 S を定義しています．:

```
S = Set(value=[1, 2, 3])
```

集合の要素が等差数列である場合は range 関数が便利です．以下の例では，自然数 1, 2, 3 を要素とする集合 S を定義しています．:

```
S = Set(value=range(1, 4))
```

添字クラス *Element* と併用することで, **変数クラス** *Variable*, **制約式クラス** *Constraint*, **定数クラス** *Parameter*, **整数変数クラス** *IntegerVariable*，*0-1* **整数変数クラス** *BinaryVariable*，**式 クラス** を集合の要素ごとに設定できます．以下の例では 3 個の変数 y[1], y[2], y[3], 3 個の定数 b[1], b[2], b[3] を定義しています．:

```
S = Set(value=[1, 2, 3])
i = Element(set=S)
y = Variable(index=i)
b = Parameter(index=i)
```

集合の要素には自然数だけでなく，文字列も使用することができます．以下の例では 2 個の整数変数 z['p'], z['q']，2 個の式 g['p'], g['q'] を定義しています．:

```
T = Set(value=['p', 'q'])
j = Element(set=T)
z = IntegerVariable(index=j)
g = 2*x[j] + 3*y[j]
```

要素の文字列は必ずしも一文字である必要はありません．:

```
T = Set(value=['before', 'after'])
```

要素の文字列が一文字である場合は直接記述することもできます．：

```
T = Set(value='pq')  # Set(value=['p', 'q']) と同じ
```

集合の要素に文字列を使用した場合は，対象を個別に記述する際に，添字部分にクォート ' またはダブルクォート " を用いる必要があります．：

```
-1 <= z['p']
```

一括して記述する場合にはクォートでまたはダブルクォートで囲んではいけません．：

```
-1 <= z[j]
```

ある集合に対して定義された添字は，その部分集合に対しても自動的に定義されます．

添字を部分集合のみ（あるいは部分集合以外）で走らせたい場合は，集合と添字の包含関係を表す演算子 <, > を利用します．以下の例では，定数 a[1], a[2] に -1 を，a[3] に 1 を設定しています．：

```
S = Set(value=[1, 2, 3])
T = Set(value=[1, 2])
i = Element(set=S)
a = Parameter(index=i)
a[i<T] = -1  # i が T に含まれる場合
a[i>T] = 1   # i が T に含まれない場合
```

多次元集合（要素の組の集合）を定義することもできます．以下の例では，二次元の添字をもつ変数 x を定義しています．各次元のすべての組み合わせについて変数を定義したいわけではない場合などに，多次元集合を使用します．：

```
IJ = Set(value=[('a', 1), ('b', 2)])
ij = Element(set=IJ)
x = Variable(index=ij)  # x['a',1] と x['b',2] が定義される
x[ij] >= 0              # x['a',1] と x['b',2] に下限を設定する
```

集合の要素数を取得するには，len 関数を使用します．以下の例では，n に集合 S の要素数を格納しています．：

```
n = len(S)
```

集合に順序がついていることを利用すると漸化式や漸化不等式を取り扱うことが可能です．

次の例では漸化不等式 $x_p \leq x_q, x_q \leq x_r, x_r \leq x_s$ を記述しています．：

```
S = Set(value=['p', 'q', 'r', 's'])
i = Element(set=S)
x = Variable(index=i)
i1 = i != S[-1]
x[i1] <= x[S.next(i1)]
```

最後の条件式 i != S[-1] は i=='s' の場合を除外するためです．集合 [index] で集合の index 番目の要素を取り出します．-1 は最後の要素を表します．上記の例では next 関数を利用しましたが，以下のように prev 関数を利用することもできます．:

```
i0 = i != S[0]
x[S.prev(i0)] <= x[i0]
```

集合の要素が整数の場合は，次のように next や prev を用いない記述も可能です．:

```
S = Set(value=[1, 2, 3, 4])
i = Element(set=S)
x = Variable(index=i)
i4 = i != 4
x[i4] <= x[i4+1]
```

同様に次の記述も可能です．:

```
i1 = i != 1
x[i1-1] <= x[i1]
```

整数以外の要素からなる集合を利用する場合，条件式において i+1, i-1 等の要素間の演算が使用できない事が，next, prev に頼らざるを得ない主な理由です．

次の例では，定数 a['p'], a['q'], a['r'] にそれぞれ 0, 2, 4（2 ずつ増加）を設定します．

```
S = Set(value=['p', 'q', 'r'])
i = Element(set=S)
a = Parameter(index=i)
a[i] = 2*S.index(i)
```

以下のように記述しても同じ意味です．:

```
S = Set(value=['p', 'q', 'r'])
i = Element(set=S)
a = Parameter(index=i)
p = Element(value=range(len(S)))
a[S[p]] = 2*p
```

集合の要素が整数である場合は，次のように index や [] を用いない記述も可能です．以下の例では，定数 a[1], a[2], a[3] にそれぞれ 2, 4, 6（2 ずつ増加）を設定します．:

```
S = Set(value=[1, 2, 3])
i = Element(set=S)
a = Parameter(index=i)
a[i] = 2*i
```

3.3.3 添字クラス Element

添字は Element というクラスで表現されます．添字とは数式 x_i の i に相当するものを意味します．添字と集合を対応させるには，キーワード引数 `set` を用います．頭文字の s は小文字である事に注意してください．添字を用いることで **変数クラス** *Variable*，**制約式クラス** *Constraint*，**定数クラス** *Parameter*, **整数変数クラス** *IntegerVariable*，*0-1* **整数変数クラス** *BinaryVariable*, **式 クラス** を要素ごとに設定できます．以下の例では，3 個の変数 y[1], y[2], y[3]，3 個の定数 b[1], b[2], b[3] を定義しています．:

```
S = Set(value=[1, 2, 3])
i = Element(set=S)
y = Variable(index=i)
b = Parameter(index=i)
```

Element の `value` キーワードを用いることにより，集合を経由せずに直接添字を定義することもできます．次の例は，上の記述と同じ意味です．:

```
i = Element(value=[1, 2, 3])
y = Variable(index=i)
b = Parameter(index=i)
```

添字のキーワード引数 `index` には Element を指定するかわりに，その Element に対応する Set を指定することもできます．次の例は，上の記述と同じ意味です．:

```
S = Set(value=[1, 2, 3])
y = Variable(index=S)
b = Parameter(index=S)
```

添字は複数導入することも可能です．次の例では 6 個の変数 x[1,'p'], x[1,'q'], x[2,'p'], x[2,'q'], x[3,'p'], x[3,'q'] を定義しています．:

```
S = Set(value=[1, 2, 3])
T = Set(value=['p', 'q'])
i = Element(set=S)
j = Element(set=T)
x = Variable(index=(i,j))
```

一つの集合に対して複数の添字を定める事もできます．次の例では 12 個の定数 a[1,'p','p'], a[1,'p','q'], a[1,'q','p'], a[1,'q','q'], a[2,'p','p'], a[2,'p','q'], a[2,'q','p'], a[2,'q','q'], a[3,'p','p'], a[3,'p','q'], a[3,'q','p'], a[3,'q','q'] を定義しています．集合 T に対して 2 つの添字 j, k が定められています．:

```
S = Set(value=[1, 2, 3])
T = Set(value=['p', 'q'])
i = Element(set=S)
j = Element(set=T)
k = Element(set=T)
a = Parameter(index=(i,j,k))
```

添字を持つ対象を個別に記述する場合は，文字列の部分のみを個別にクォート ' またはダブルクォート " で囲む必要があります．:

```
x[1,'p','q']
```

次の例はいずれも誤りです．:

```
x['1','p','q']
x[1,'p,q']
```

以下のようにクォートで囲まないと添字は自動展開され，一括して扱われます．（**添字の自動展開機能**）:

```
x[i,j,k] >= 1
```

添字は，属する集合が整数値を取る場合には次のような演算子を用いることができます．

+ （加算）	- （減算）	* （乗算）	% （剰余）
/ （浮動小数除算）	// （整数除算）	** （冪）	

次の例では，定数 a[1], a[2], a[3] に値 2, 4, 6 を設定しています．:

```
i = Element(value=[1, 2, 3])
a = Parameter(index=i)
a[i] = 2*i
```

次の例では，制約式 $x_2 + x_4 + x_6 \leq 5$ を記述しています．制約式の左辺を定義するために偶数番目の項のみの和を取得しています．:

```
i = Element(value=[1, 2, 3, 4, 5, 6])
x = Variable(index=i)
ieven = i%2==0
Sum(x[ieven], ieven) <= 5
```

次の例では，漸化不等式 $x_i \leq x_{i+1}$ を定義しています．:

```
i = Element(value=[1, 2, 3, 4])
x = Variable(index=i)
i4 = i!=4
x[i4] <= x[i4+1]
```

3.3.4 定数クラス Parameter

定数は Parameter というクラスで表現されます．具体的に a という定数を定義するには以下のように記述します．:

```
a = Parameter()
```

複数の定数を一度に定義するには，index キーワードと添字クラス Element を用います．以下の例では，3 個の定数 b[1], b[2], b[3] を一度に定義しています．:

```
i = Element(value=[1, 2, 3])
b = Parameter(index=i)
```

以下の例では，6 個の定数 c[1,'p'], c[1,'q'], c[2,'p'], c[2,'q'], c[3,'p'], c[3,'q'] を一度に定義しています．:

```
i = Element(value=[1, 2, 3])
j = Element(value=['p', 'q'])
c = Parameter(index=(i,j))
```

定数の値は = で設定します．定数の値を明示的に指定しない場合は，自動的に 0 が設定されます．以下の例では定数 b[1], b[2], b[3] に 5 をまとめて設定しています．:

```
b[i] = 5
```

value キーワードを用いて宣言と同時に値を設定することもできます．:

```
b = Parameter(index=i, value=5)
```

宣言時に個別に値を設定する場合は辞書を使います．以下の二表現は同様の意味を持ちます．:

```
b = Parameter(index=i)
b[1] = 10
b[2] = 20
b[3] = 30
```

```
b = Parameter(index=i, value={1: 10, 2: 20, 3: 30})
```

複数の引数を持たせる場合，順序は任意です．以下の二表現は同様の意味を持ちます．:

```
b = Parameter(index=i, value=5)
```

```
b = Parameter(value=5, index=i)
```

添字がない定数の場合は後から値を変えることはできません．以下ではオブジェクト a は定数型の後に Python の int 型になってしまいます．:

```
a = Parameter()
a = 1  # NG. a = Parameter(value=1) とすること
```

3.3.5 変数クラス Variable

変数は Variable というクラスで表現されます．具体的に x という変数を定義するには以下のように記述します．:

```
x = Variable()
```

複数の変数を一度に定義するには，index キーワードと添字クラス Element を用います．以下の例では，3 個の変数 y[1], y[2], y[3] を一度に定義しています．:

```
i = Element(value=[1, 2, 3])
y = Variable(index=i)
```

変数の初期値は = で設定できます．以下の例では y[1], y[2], y[3] に初期値 5 をまとめて設定しています．:

```
y[i] = 5
```

init キーワードを用いて宣言と同時に初期値を設定することもできます．:

```
y = Variable(index=i, init=5)
```

宣言時に個別に値を設定する場合は辞書を使います．以下の二表現は同様の意味を持ちます．:

```
y = Variable(index=i)
y[1] = 10
y[2] = 20
y[3] = 30
```

```
y = Variable(index=i, init={1: 10, 2: 20, 3: 30})
```

明示的な指定が無い場合，変数の初期値はアルゴリズムに応じて自動的に決定されます．アルゴリズムによっては初期値の設定を無視します．

複数の引数を持たせる場合，順序は任意です．以下の二表現は同様の意味を持ちます．:

```
y = Variable(index=i, init=5)
```

```
y = Variable(init=5, index=i)
```

lb キーワード引数, ub キーワード引数を用いて宣言時に変数の下限値，上限値を設定することもできます．:

```
z = Variable(index=i, lb=0, ub={1: 5, 2: 3, 3: 4})
```

3.3.6 式 クラス

式は演算結果を代入によって保持することで利用できます．例えば，2x + 3y という式を定義したい場合，次のように記述します．:

```
x = Variable()
y = Variable()
g = 2*x + 3*y
```

以下の例では，3 個の式 g[1], g[2], g[3] を一度に定義しています．:

```
i = Element(value=[1, 2, 3])
x = Variable(index=i)
y = Variable(index=i)
g = 2*x[i] + 3*y[i]
```

式の宣言では左辺に括弧は不要です．以下の記述は誤りです．:

```
g[i] = 2*x[i] + 3*y[i]
```

式に明示的に名前を付けるには以下のようにします．:

```
g = 2*x[i] + 3*y[i]
g.name = 'g'
```

上のように定義した式を使用する場合は g または g[i] とします．式はあくまで置き換えであるので以下の表現はいずれも同様の意味を持ちます．:

```
f = 2*x[i] + 3*y[i] + 4
```

```
g = 2*x[i] + 3*y[i]
f = g + 4
```

```
g = 2*x[i] + 3*y[i]
f = g[i] + 4
```

式を使う事によりモデルの記述を簡略化することができます．同じ式が何度も出現するモデルにおいて特に有効です．

C++SIMPLE のように式の宣言はできません．

3.3.7 制約式クラス Constraint

制約式は Constraint というクラスで表現されます．PySIMPLE で表現可能な制約式は，等式制約（== を使用）及び等号付不等式制約（<=, >= を使用）の二種類です．**等式制約に用いる演算子は=ではなく，==であることに注意してください．**具体的に $x + y = 1$ という制約式を定義するには次のように記述します．

```
x + y == 1
```

$x - 2y \leq 0$ という制約式を定義するには次のように記述します．：

```
x - 2*y <= 0
```

等号の付かない不等式を取り扱う事はできません．次の記述は誤りです．：

```
x - 2*y < 0
```

複数の制約式を一度に定義するには, 添字クラス Element を用います．以下の例では, 3 個の制約式 $x_1 - 2y_1 \leq 0, x_2 - 2y_2 \leq 0, x_3 - 2y_3 \leq 0$ を一度に定義しています．：

```
i = Element(value=[1, 2, 3])
x = Variable(index=i)
y = Variable(index=i)
x[i] - 2*y[i] <= 0
```

不一致制約を表す演算子 != を用いることはできません．以下の記述は誤りです．：

```
x + y != -1
```

3.3.8 制約式種

全ての制約式は次の 3 つの制約式種に分類されます．

- ハード制約 (通常)
- セミハード制約
- ソフト制約

ハード制約とは，最も優先して満たすべき制約式（必ず満たさなければならない制約式）のことです．制約式種を指定しない場合はハード制約となります．

セミハード制約とは，ハード制約の次に優先して満たすべき制約式のことです．通常，必ず満たさなければならない制約式の一部をセミハード制約とし，実行不可能性の原因をセミハード制約に押し付けるという使い方をします．

セミハード制約は wcsp のみで有効です．他のアルゴリズムにおいてはセミハード制約はエラーあるいはハード制約として解釈されます．

ソフト制約は，優先度がもっとも低く，必ずしも満たす必要はないが，できるだけ満たして欲しい制約式のことです．その際，ソフト制約は，各制約式の違反量からペナルティ量を計算し，その総ペナルティ量が最も小さくなることをもってできるだけ制約式を満たしたと解釈します (計算方法は後述).

ソフト制約は wcsp/wls/simplex/asqp のみで有効です．他のアルゴリズムにおいてはソフト制約はハード制約として解釈されます．

問題クラス・解法と制約種の対応は次のようになります．

制約式種	wcsp	wls	wcsp/wls 以外
ハード制約	○	○	○
セミハード制約	○	○※ 1	△※ 1 ※ 2
ソフト制約	○	○	△※ 2

※ 1 HardConstraint 扱い※ 2 一部の解法のみ

目的関数の値は SoftConstraint(1) のソフト制約と同等に扱われます．目的関数の値が負の場合はペナルティ量は 0 となります．

ハード制約関数 HardConstraint

制約がハード制約であることを指定するには制約式に続いてカンマで区切って HardConstraint 関数を用います．:

```
problem += z[i] >= 1, HardConstraint()
```

制約式種を指定しない場合，ハード制約と認識されるため，以下の表現はいずれも同様の意味を持ちます．:

```
problem += z[i] >= 1
problem += z[i] >= 1, HardConstraint()
problem += z[i] >= 1, SoftConstraint(-1)  # 後述
```

また，制約式に名前を付ける場合，制約式種と名前の順序は任意です．以下の表現はいずれも同様の意味を持ちます．:

```
problem += z[i] >= 1, 'cons'
problem += z[i] >= 1, HardConstraint(), 'cons'
problem += z[i] >= 1, 'cons', HardConstraint()
```

ハード制約ではペナルティ量 = 違反量です．

セミハード制約関数 SemiHardConstraint

制約がセミハード制約であることを指定するには制約式に続いてカンマで区切って SemiHardConstraint 関数を用います．:

```
problem += z[i] >= 1, SemiHardConstraint()
problem += z[i] >= 1, SoftConstraint(-2)  # 同じ．後述
```

セミハード制約ではペナルティ量 = 違反量です．

ソフト制約関数 SoftConstraint

制約がソフト制約であることを指定するには制約式に続いてカンマで区切って SoftConstraint 関数を用います．:

```
problem += z[i] >= 1, SoftConstraint(1)
```

SoftConstraint 関数は引数により制約式の違反量の計算パラメータの設定もできます．:

```
def SoftConstraint(weight: int, quad=None: float, linear=None: float):
```

- **weight: 重みの基本となる値．非負整数または -1 または -2．** -1, -2 はそれぞれ HardConstraint, SemiHardConstraint を表す
- quad: 重みの二次の係数．非負値．
- linear: 重みの一次の係数．非負値．

制約式 cons が制約を満たしていない (cons.violation>0) の場合，ソフト制約のペナルティ量 penalty は以下で定義されます．:

```
v = cons.violation
penalty = (quad*v**2 + linear*v) * weight  # wcsp
penalty =                        v  * weight  # wcsp 以外
```

また，SoftConstraint 関数の第 2 引数と第 3 引数は省略することができ，次のような規則により解釈されます．:

```
# 第 2, 3 引数の省略
SoftConstraint(weight)       == SoftConstraint(weight, 0, 1)     # penalty = v*weight
と等価
# 第 3 引数の省略
SoftConstraint(weight, quad) == SoftConstraint(weight, quad, 0)  # penalty =␣
↪quad*v**2 * weight と等価
```

このペナルティ値は weight 属性と violation 属性の積で計算できます．:

```
z = BinaryVariable()
problem = Problem()
problem += z >= 1, 'cons1', SoftConstraint(3)        # penalty = 1 * 3
problem += z >= 1, 'cons2', SoftConstraint(3, 2)     # penalty = 2*1**2 * 3
problem += z >= 1, 'cons3', SoftConstraint(3, 2, 1)  # penalty = (2*1**2 + 1*1) * 3
for cons in problem.constraints.values():
    print(cons.weight*cons.violation)
```

3.3.9 整数変数クラス IntegerVariable

整数変数は IntegerVariable というクラスで表現されます．具体的に x という整数変数を定義するには以下のように記述します．:

```
x = IntegerVariable()
```

以下のように書いても同じです．:

```
x = Variable(type=int)
```

複数の整数変数を一度に定義するには，添字クラス Element を用います．以下の例では，3 個の整数変数 y[1], y[2], y[3] を一度に定義しています．:

```
i = Element(value=[1, 2, 3])
y = IntegerVariable(index=i)
```

複数の引数を持たせる場合，順序は任意です．以下の二表現は同様の意味を持ちます．:

```
z = Variable(type=int, index=i)
```

```
z = Variable(index=i, type=int)
```

3.3.10 0-1 整数変数クラス BinaryVariable

0-1 整数変数は BinaryVariable というクラスで表現されます．具体的に x という整数変数を定義するには以下のように記述します．:

```
x = BinaryVariable()
```

以下のように書いても同じです．:

```
x = Variable(type=bin)
```

複数の 0-1 整数変数を一度に定義するには，添字クラス Element を用います．以下の例では，3 個の 0-1 整数変数 y[1], y[2], y[3] を一度に定義しています．:

```
i = Element(value=[1, 2, 3])
y = BinaryVariable(index=i)
```

複数の引数を持たせる場合，順序は任意です．以下の二表現は同様の意味を持ちます．:

```
z = Variable(type=bin, index=i)
```

```
z = Variable(index=i, type=bin)
```

3.3.11 範囲演算関数 Sum, Prod

数式における $\sum$ や $\prod$ に類する機能として，PySIMPLE では範囲演算関数として，Sum 関数と Prod 関数が提供されています．次の例では，制約式 $\sum_{i=1}^{3} x_i = 10$ を記述しています．:

```
i = Element(value=[1, 2, 3])
x = Variable(index=i)
Sum(x[i], i) == 10
```

上の記述を Sum 関数を使わずに書くと次のようになります．:

```
i = Element(value=[1, 2, 3])
x = Variable(index=i)
x[1] + x[2] + x[3] == 10
```

次の例では，制約式 $(\prod_{i=1}^{3} a_i)x = 20$ を記述しています．:

```
i = Element(value=[1, 2, 3])
a = Parameter(index=i)
x = Variable()
Prod(a[i], i)*x == 20
```

上の記述を Prod 関数を使わずに書くと次のようになります．:

```
i = Element(value=[1, 2, 3])
a = Parameter(index=i)
x = Variable()
a[1]*a[2]*a[3]*x == 20
```

複数の添字に対して適用する事もできます．次の例では，制約式 $\sum_{i=1}^{3}\sum_{j=1}^{2} a_i b_j y_{ij} = 10$ を記述しています．:

```
i = Element(value=[1, 2, 3])
j = Element(value=[1, 2])
y = Variable(index=(i,j))
a = Parameter(index=i)
```

(次のページに続く)

(前のページからの続き)

```
b = Parameter(index=j)
Sum(a[i]*b[j]*y[i,j], (i,j)) == 10
```

次のように記述することも可能です．:

```
Sum(Sum(a[i]*b[j]*y[i,j], j), i) == 10
```

すべての添字について演算を行う場合は第二引数を省略することもできます．:

```
Sum(a[i]*b[j]*y[i,j]) == 10
```

次の $\sum_{i=1}^{3} a_i b_j y_{ij} = 10$ のように一部の添字について演算を行う場合は第二引数を省略することはできません．:

```
Sum(a[i]*b[j]*y[i,j], i) == 10
```

条件式を用いて，和や積を取る範囲を制限する事もできます．次の例では，制約式 $\sum_{i=3}^{5} x_i = 10$ を記述しています．:

```
i = Element(value=range(1, 6))
x = Variable(index=i)
i3 = i>=3
Sum(x[i3], i3) == 10
```

条件付けられた添字と条件付けられていない添字を混在することはできません．次の例はいずれも誤りです．:

```
Sum(x[i], i>=3) == 10
Sum(x[i>=3], i) == 10
```

条件式が複数箇所に登場する場合，条件式は一度保存したものを使用する必要があります．次の例は誤りです．:

```
Sum(x[i>=3], i>=3) == 10
```

条件付けられた添字が一度しか登場しない場合は直接記述することができます．:

```
Sum(x[i>=3]) == 10
```

次の例では，制約式 $\sum_{i \in T} x_i = 10$，$\sum_{i \notin T} x_i = 20$ を記述しています．:

```
i = Element(value=['p', 'q', 'r', 's'])
T = Set(value=['p', 'r'])
x = Variable(index=i)
iinT = i<T
Sum(x[iinT], iinT) == 10
```

(次のページに続く)

(前のページからの続き)

```
inotinT = i>T
Sum(x[inotinT], inotinT) == 20
```

次のように記述することもできます．:

```
i = Element(value=['p', 'q', 'r', 's'])
T = Set(value=['p', 'r'])
x = Variable(index=i)
Sum(x[i<T]) == 10
Sum(x[i>T]) == 20
```

集合を用意しなくても記述することができます．次の例はいずれも正しい書き方です．:

```
i = Element(value=['p', 'q', 'r', 's'])
x = Variable(index=i)
iinT = i<['p', 'r']
Sum(x[iinT], iinT) == 10
inotinT = i>['p', 'r']
Sum(x[inotinT], inotinT) == 20
```

```
i = Element(value=['p', 'q', 'r', 's'])
x = Variable(index=i)
Sum(x[i<['p', 'r']]) == 10
Sum(x[i>['p', 'r']]) == 20
```

3.3.12 選択関数 Selection

選択関数 Selection は，添字つき 0-1 整数変数の中で一つだけを 1 に固定したい場合に用います．同様の記述は Sum 関数を用いる事でも可能ですが，**制約充足問題ソルバ** *wcsp/wls* 利用時には利点があります．wcsp を利用する際には，内部的には複数の 0-1 整数変数を用意する替わりに一つの離散変数を用意するため，内部処理が高速化されます．wls を利用する際は Selection 関数を用いると内部で探索戦略が変更され，より効率的な探索を行います．wcsp/wls 以外ではハード制約の Sum(…)==1 と等価です．

次の例では，3 つ の 0-1 整数変数 z[1], z[2], z[3] のうち一つだけを 1 にするよう指定しています．:

```
i = Element(value=[1, 2, 3])
z = BinaryVariable(index=i)
problem = Problem()
problem += Selection(z[i], i)
```

すべての添字について演算を行う場合は第二引数の添字は省略することもできます．:

```
problem += Selection(z[i])
```

Sum 関数を利用した場合，次のようになります．:

```
problem += Sum(z[i], i) == 1
```

Selection 関数の引数には，条件式を指定することもできます．次の例では，z[1], z[2] のうち一つだけを 1 にするよう指定しています．:

```
p += Selection(z[i<=2])
```

範囲指定の方法については **範囲演算関数** *Sum, Prod* も参考にしてください．

Selection を用いた制約に明示的に制約式種を指定することはできませんが，常に HardConstraint 扱いとなります．

3.3.13 条件式

条件式は，添字の動く範囲を制限する機能を有します．次の例では，定数 a[1], a[2] に -1 を，a[3] に 1 を設定しています．:

```
i = Element(value=[1, 2, 3])
a = Parameter(index=i)
a[i<=2] = -1
a[i>=3] = 1
```

条件式が一つの式の複数箇所に登場する場合，条件式は一度保存したものを使用する必要があります．

```
i = Element(value=[1, 2, 3])
a = Parameter(index=i)
x = Variable(index=i)
i2 = i<=2
ax = a[i2] + x[i2]
```

次の例は誤りです．:

```
ax = a[i<=2] + x[i<=2]
```

条件付けられた添字と条件付けられていない添字を混在することはできません． 次の例はいずれも誤りです．:

```
ax = a[i] + x[i<=2]
ax = a[i<=2] + x[i]
```

条件式には等号 ==，等式付不等号 <= >=，不等号 < >，不一致 != 演算子が使用できます（制約式に使用できるのは，等号と等式付不等号のみです）．次の例では，定数 asum の値を定数 a[i] の中で 0 より大きいものの和 $\sum_{a_i>0} a_i$ で定めています．:

```
i = Element(value=[1, 2, 3])
a = Parameter(index=i, value={1: 10, 2: -20, 3: 30})
```

(次のページに続く)

(前のページからの続き)

```
apos = a[i]>0
asum = Sum(a[apos], apos)
```

上の例では，Sum 関数の第二引数を省略することで一度に記述することができます．:

```
asum = Sum(a[a[i]>0])
```

不等号 < > は添字に対する集合の所属有無を表現する演算子としても使用されます．以下の例では，定数 a[1], a[2] に -1 を，a[3] に 1 を設定しています．:

```
i = Element(value=[1, 2, 3])
T = Set(value=[1, 2])
a = Parameter(index=i)
a[i<T] = -1  # i が T に含まれる場合
a[i>T] = 1   # i が T に含まれない場合
```

集合を用いずに直接記述することもできます．:

```
i = Element(value=[1, 2, 3])
a = Parameter(index=i)
a[i<[1, 2]] = -1  # i が [1, 2] に含まれる場合
a[i>[1, 2]] = 1   # i が [1, 2] に含まれない場合
```

条件式同士を演算子 &, | で連結することができます．それぞれ，and, or を意味します．次の例では，定数 b[1], b[4] に -1 を，b[2], b[3] に 1 を設定しています．:

```
i = Element(value=[1, 2, 3, 4])
b = Parameter(index=i)
b[(i<=1) | (i>=4)] = -1
b[(i>=2) & (i<=3)] = 1
```

演算子の優先順位により，それぞれの条件式を括弧でくくる必要があることに注意してください．

多次元の条件式を作ることもできます．:

```
i = Element(value=[1, 2, 3])
j = Element(value=[1, 2, 3])
x = Variable(index=(i,j))
print(x[i<j])
```

この出力は次のようになります．:

```
x:
x[1,2]
x[1,3]
x[2,3]
```

多次元の条件式の一部の次元を用いる場合は () で必要な次元だけを取り出す必要があります．次の例では二次元の条件式 i<j の 1 次元目を使用して変数 y にアクセスしています．:

```
i = Element(value=[1, 2, 3])
j = Element(value=[1, 2, 3])
x = Variable(index=(i,j))
y = Variable(index=i)
ij = i<j
print(x[ij] + y[ij(0)])
```

次元は 0 から数えます．この出力は次のようになります．:

```
(x[(i<j)]+y[(i<j)(0)]):
x[1,2]+y[1]
x[1,3]+y[1]
x[2,3]+y[2]
```

条件式に変数や式を用いることはできません．次の記述は誤りです．:

```
i = Element(value=[1, 2])
x = Variable(index=i)
y = Variable(index=i)
xpos = x[i] >= 0
3*x[xpos] + 2*y[xpos] == 0
```

Condition 関数

複雑な条件式を作成するために Condition 関数を用いることができます．次の記述はいずれも同じ意味を持ちます．:

```
i = Element(value=[1, 2, 3])
i1 = i>1
```

```
i = Element(value=[1, 2, 3])
i1 = Condition(i, i>1)
```

次の例では二次元集合 IJ に対して一次元目が 1 である条件式を作成しています．:

```
ij = Element(value=[(1,3), (1,4), (2,3)])
ij0 = ij(0)==1
```

ここで注意すべきは ij0 は一次元の条件式となるので，ij0 は 1 の範囲のみを動きます．上の例は以下と同じ意味を持ちます．:

```
ij = Element(value=[(1,3), (1,4), (2,3)])
ij0 = Condition(ij(0), ij(0)==1)
```

二次元の条件式を作成するには次のようにします．：

```
ij = Element(value=[(1,3), (1,4), (2,3)])
ij0 = Condition(ij, ij(0)==1)
```

ここで ij0 は二次元の条件式なので，範囲 (1,3) と (1,4) を動くことになります．

第二引数に条件式のタプルを与えることによって複数の条件を考慮することもできます．：

```
i = Element(value=[1, 2, 3])
i2 = Condition(i, (1<i, i<3))
```

```
i = Element(value=[1, 2, 3])
j = Element(value=[4, 5])
ij = Condition((i,j), (i!=2, j>4))
```

このように Condition 関数を用いることによって複雑な条件式を作成することができるようになります．

3.3.14 範囲最小（大）値取得関数 Min，Max

範囲最小値取得関数 Min 及び範囲最大値取得関数 Max は，添字付けされた定数式の中から最小（大）のものを返す関数です．：

```
i = Element(value=[1, 2])
j = Element(value=['p', 'q'])
avalue = {(1,'p'): 10, (1,'q'): 11, (2,'p'): 20, (2,'q'): 21}
a = Parameter(index=(i,j), value=avalue)
Min(a[i,j], (i,j))  # 10
```

Sum 関数と同様にすべての範囲に亘る場合は第二引数を省略することができます．：

```
Min(a[i,j])  # 10
```

一部の添字について演算を行う場合は 第二引数を省略することはできません．次の例は $\min_i a_{i,j}$ と $\min_j a_{i,j}$ を表しています．：

```
Min(a[i,j], i)
# Min(a[i,j], i)['X']=10
# Min(a[i,j], i)['Y']=11
Min(a[i,j], j)
# Min(a[i,j], j)[1]=10
# Min(a[i,j], j)[2]=20
```

条件式を用いて，添字の範囲を制限する事もできます． 次の例では，$\min_{i>3} a_i$ を記述しています．：

```
i = Element(value=range(1, 6))
a = Parameter(index=i)
a[i] = i
Min(a[i>3])  # 4
```

その他の例は **範囲演算関数** *Sum, Prod* も確認してください.

最小（大）値取得関数 MinOf，MaxOf

最小値取得関数 MinOf，最大値取得関数 MaxOf は各最小（大）のものを返す関数です．:

```
i = Element(value=[11, 21, 31])
a = Parameter(index=i, value={11: 22, 21: 32, 31: 12})
b = Parameter(index=i, value={11: 33, 21: 13, 31: 23})
Printf('i={i}, a={a}, b={b}', i=i, a=a[i], b=b[i])
# i=11, a=22, b=33
# i=21, a=32, b=13
# i=31, a=12, b=23
MinOf(i, a[i], b[i])
# MinOf(i, a[i], b[i])[1]=11
# MinOf(i, a[i], b[i])[2]=13
# MinOf(i, a[i], b[i])[3]=12
```

次の例は MinOf 関数を用いて変数の上限を一度に設定しています．:

```
x = Variable(index=i)
x[i] <= MinOf(i, a[i], b[i])
```

3.3.15 数学関数

PySIMPLE では次の演算と数学関数が定義されています．それぞれの意味はプログラミング言語 Python におけるものと同じです.

+	-	*	%	/	//	**
Ceil	Floor	Fabs	Fmod			
Exp	Log	Log10	Sqrt			
Sin	Cos	Tan	Asin	Acos	Atan	Atan2
Sinh	Cosh	Tanh	Asinh	Acosh	Atanh	
Hypot	Erf					

次の例では，制約式 $a_i^3 + x_i \leq 11$ を記述しています．:

```
i = Element(value=[1, 2, 3])
a = Parameter(index=i)
x = Variable(index=i)
a[i]**3 + x[i] <= 11
```

変数の累乗は一次，または二次のみを扱うことができます．以下の表現はいずれも同様の意味を持ちます．:

```
a[i] + x[i]*x[i] <= 11
a[i] + x[i]**2 <= 11
a[i] + pow(x[i], 2) <= 11
```

3.4 制約充足問題ソルバ wcsp/wls

制約充足問題とは制約条件を満たす解を見つける組合せ最適化問題です．PySIMPLE では制約充足問題に対するメタヒューリスティクスアルゴリズムとして，wcsp(weighted constraint satisfaction problem)，wls(weighting local search) を用いることができます．制約充足問題ソルバを用いる場合は他のアルゴリズムと異なり近似解法であり，厳密解である保証がありません．

wcsp と wls の違いは以下のとおりです．

	wcsp	wls
探索方法	タブー探索	局所探索
近傍操作	一度に最大 2 つの変数の値を変更	一度に最大 4 つの変数の値を変更
探索傾向	実行可能解の早期発見を目的とした探索	目的関数・ソフト制約を重視した上で実行可能解を探索
連続変数	非対応	対応[1]
添字グループ	非対応	割当を表す変数への指定により探索範囲を拡大[2]
目的関数	目標値を用いたソフト制約化	目的関数のまま扱う
Selection 制約	充足を保ちながら探索	制約式へ変換，探索戦略を修正
セミハード制約	対応	非対応（ハード制約へ変換）
制約違反量	整数値へ変換（小数点以下切捨て）[3]	実数値のまま扱う
C++SIMPLE	対応	非対応
得意な問題例	選択・割当を表す変数が多い問題． 特に，Selection を多く含むモデル．	選択・割当を表す変数が多い問題． 特に，係数が 0-1 の制約式が多い問題（集合被覆制約など）．また，充足の難しい制約式をもつ問題（集合分割制約・等号ナップサック制約など）．

[1] 線形の目的関数・制約式のみ対応しています．
[2] 詳細は *Variable.group* をご覧ください．
[3] wcsp/wls は制約式や目的関数をペナルティとして認識し，ペナルティが小さくなる答えを探すアルゴリズムです．Nuorium Optimizer の内部でこのペナルティを評価する際に小数点以下を切り捨て，整数として評価を行います．よって，「0 >= 1」のような制約式は制約違反（ペナルティ 1）となりますが，「0 >= 0.99」というような制約式は制約を満たしている（ペナルティ 0）となります．定式化の段階で小数点以下も考慮する必要がある目的関数や制約式に関しては，該当する式（制約式ならば両辺）に大きな値（1000 ならば小数第三桁まで有効になる）をかけておく必要があります．ただし値が極端に大きくならないよう注意してください．

制約充足問題ソルバではハード制約，セミハード制約，ソフト制約といった制約式種の活用が有効です．詳細は 制約式種 をご覧ください．

3.5 出力制御

PySIMPLE で記述された数理計画モデルは，Nuorium Optimizer で求解されます．その際には求解情報が標準出力に，(出力指定があれば) より細かい解情報が解ファイル（モデル名.sol）に出力されます．

この章では，出力情報の追加に用いられる PySIMPLE の関数 Printf, Fprintf ついて説明します．

3.5.1 出力対象

後述する Printf 関数，Fprintf 関数では，以下の構成要素に対する情報を適宜取得することができます．

構成要素	情報	意味
Variable	val	現在値
	init	初期値
	dual	双対変数値
	ub	上限値
	lb	下限値
Expression	val	現在値
	init	初期値
Constraint	dual	双対変数値
	violation	違反量

IntegerVariable と BinaryVariable は Variable の糖衣構文であるので Variable と同じです．

現在値 val は solve 関数が呼ばれる前であれば初期値，呼ばれた後には最適化計算結果が入っています．例えば，次のモデルに対する出力は以下のようになります．:

```
p = Problem(type=min)
i = Element(value=[1, 2, 3])
x = Variable(index=i)
p += Sum(2*x[i])
p += x[i] >= 5
x[i] = 10
print(x.val)  # 10
p.solve(silent=True)
print(x.val)  # 5
```

出力:

```
x[1].val=10
x[2].val=10
```

(次のページに続く)

(前のページからの続き)

```
x[3].val=10
x[1].val=5.0000000020802062
x[2].val=5.0000000020802062
x[3].val=5.0000000020802062
```

3.5.2 print 関数

print 関数は，Python の組み込み関数です．PySIMPLE では print 関数により PySIMPLE のさまざまなオブジェクトに関する情報を，決まったフォーマットで出力させる機能を有しています．

print 関数を用いてオブジェクトの情報を出力するには以下のように記述します．

```
print(出力対象)
```

集合の値を出力するには，次のように記述します．:

```
S = Set(value=[1, 2, 3])
print(S)
```

定数の値を出力するには，次のように記述します．:

```
i = Element(value=[1, 2, 3])
a = Parameter(index=i)
print(a)
print(a[i])
```

変数を出力するには，次のように記述します．:

```
i = Element(value=[1, 2, 3])
x = Variable(index=i)
print(x)
print(x[i])
```

変数の現在値を出力するには，次のように記述します．:

```
print(x.val)
print(x[i].val)
```

整数変数の下限値を出力するには，次のように記述します．:

```
z = IntegerVariable(index=i, lb=0)
print(z.lb)
print(z[i].lb)
```

式の初期値を出力するには，次のように記述します．:

```
g = a[i] + x[i]
print(g.init)
print(g[i].init)
```

制約式の双対変数値を出力するには，次のように記述します．:

```
cons = x[i] >= a[i]
print(cons.dual)
print(cons[i].dual)
```

問題に登録された情報を出力するには，次のように記述します．:

```
p = Problem()
x = Variable()
p += 2*x      # 目的関数
p += x >= 0   # 制約式
print(p)
```

求解関数 solve の前に print 関数を記述すると，求解前の初期状態の情報が記述されます．例えば，次のモデルに対する出力は以下のようになります．:

```
p = Problem(type=min)
i = Element(value=[1, 2, 3])
x = Variable(index=i)
p += Sum(2*x[i])
p += x[i] >= 5
x[i] = 10
print(x.val)  # 10
p.solve(silent=True)
print(x.val)  # 5
```

出力:

```
x[1].val=10
x[2].val=10
x[3].val=10
x[1].val=5.000000020802062
x[2].val=5.000000020802062
x[3].val=5.000000020802062
```

次の例は求解結果を表す要素 result の情報を出力させています．一覧は *pysimple.problem.Result* を確認してください．:

```
x = Variable()
y = Variable()
p = Problem()
p += 2*x + 3*y
```

(次のページに続く)

(前のページからの続き)

```
p += x + 2*y == 15
p += x >= 0
p += y >= 0
p.solve(silent=True)
print(p.result)
```

出力:

```
errorMessage          : ''
method                : 'higher'
optValue              : 22.50000000376254
errorCode             : 0
nvars                 : 2
nfunc                 : 4
iters                 : 6
elapseTime            : 0.0020000040531158447
peakPhysicalMemoryUsed: 44
peakVirtualMemoryUsed : 1518
fevals                : 9
factCount             : 7
tolerance             : 1e-08
residual              : 3.762538162000488e-09
infeasibility         : 3.552713678800501e-15
consInfeasibility     : 0.0
varInfeasibility      : 0.0
hardPenalty           : 0.0
semiHardPenalty       : 0.0
softPenalty           : 0.0
IIS                   : no IIS
```

出力範囲を，条件式で制限する事も可能です．以下のようにした場合，変数 x[1], x[2] の現在値のみが出力されます．:

```
print(x[i<3].val)
```

正の値を持つ添字のみの現在値を出力するには次のようにします．:

```
print(x[x[i].val>0].val)
```

要素や値が文字列の場合，print 関数だけでは引用符は表示されません．引用符まで表示させるには repr 関数を用います．:

```
j = Element(value=['p', 'q'])
a = Parameter(index=j, value={'p': 'pair', 'q': 'queue'})
print(a)
print(repr(a))
```

この出力は次のようになります．：

```
a[p]=pair
a[q]=queue
a['p']='pair'
a['q']='queue'
```

インタプリタでは repr の表示がデフォルトとなります．：

```
>>> j = Element(value=['p', 'q'], name='i')
>>> a = Parameter(index=j, value={'p': 'pair', 'q': 'queue'}, name='a')
>>> a
a['p']='pair'
a['q']='queue'
>>> print(a)
a[p]=pair
a[q]=queue
```

添字なしの一部オブジェクトにはフォーマット指定が可能です．：

```
a = Parameter(value=3.14)
x = Variable(init=2.72)
print(f'{a} {a:.1f} {x.val} {x.val:.1f}')  # a=3.14 a=3.1 x.val=2.72 x.val=2.7
```

3.5.3 Printf 関数

Printf 関数は PySIMPLE の構成要素の情報を任意のフォーマットで出力させる機能を有しています．Printf 関数の書式は以下のように定められています．：

```
Printf(出力指定書式，出力対象 1，出力対象 2，..)
```

次の例では，変数の現在値を出力させています．：

```
i = Element(value=[1, 2])
x = Variable(index=i, init={1: 10, 2: 20})
Printf('{}', x[i].val)
```

これに対する出力は以下のようになります．：

```
10
20
```

次のように記述すると，出力は以下のようになります．：

```
Printf('x[{}] の値 = {}', i, x[i].val)
```

出力:

```
x[1] の値 = 10
x[2] の値 = 20
```

変数や式に対して .val の有無は意味が異なります．：

```
Printf('{}', x[i])
```

この出力は次のようになります．：

```
x[1]
x[2]
```

以下は小数を出力する例です．小数を出力するには {:f} を用います．：

```
Printf('x[{}] の値 = {:f}', i, x[i].val)
```

出力:

```
x[1] の値 = 10.000000
x[2] の値 = 20.000000
```

表示させる桁数を指定することもできます．以下の例では，小数点以下二桁のみが出力されるよう記述しています．：

```
Printf('x[{}] の値 = {:.2f}', i, x[i].val)
```

出力:

```
x[1] の値 = 10.00
x[2] の値 = 20.00
```

出力の幅を指定することもできます．以下の例では，半角 15 文字に出力が収まるように記述しています．：

```
Printf('x[{}] の値 = {:15f}', i, x[i].val)
```

出力:

```
x[1] の値 =       10.000000
x[2] の値 =       20.000000
```

桁数と出力幅の両方をまとめて記述することもできます．：

```
Printf('x[{}] の値 = {:15.2f}', i, x[i].val)
```

出力:

```
x[1] の値 =           10.00
x[2] の値 =           20.00
```

フォーマットには他にもさまざまな書式を使用することができます．詳細は *pysimple.Printf* を確認してください．

求解関数 solve の前に Print 関数を記述すると，求解前の初期状態の情報が記述されます．例えば，次のモデルに対する出力は以下のようになります．

```
p = Problem(type=min)
i = Element(value=[1, 2, 3])
x = Variable(index=i)
p += Sum(2*x[i])
p += x[i] >= 5
x[i] = 10
Printf('x[{}] の値 = {:f}', i, x[i].val)  # 10
p.solve(silent=True)
Printf('x[{}] の値 = {:f}', i, x[i].val)  # 5
```

出力:

```
x[1] の値 = 10.000000
x[2] の値 = 10.000000
x[3] の値 = 10.000000
x[1] の値 = 5.000000
x[2] の値 = 5.000000
x[3] の値 = 5.000000
```

出力範囲を，条件式で制限する事も可能です．以下のようにした場合，変数 x[1], x[2] の値のみが出力されます．:

```
i3 = i<3
Printf('x[{}] の値 = {:f}', i3, x[i3].val)
```

同じ添字の対象であれば，同時に複数出力することができます．次の例では，変数 x[1], x[2], x[3]，定数 a[1], a[2], a[3] の値を同時に出力させています．:

```
i = Element(value=[1, 2, 3])
x = Variable(index=i, init=3)
a = Parameter(index=i, value=5)
Printf('x[{:d}] = {:f}, a[{:d}] = {:f}', i, x[i].val, i, a[i])
```

出力:

```
x[1] = 3.000000, a[1] = 5.000000
x[2] = 3.000000, a[2] = 5.000000
x[3] = 3.000000, a[3] = 5.000000
```

次の例は求解結果を表す要素 result の情報を出力させています．一覧は *pysimple.problem.Result* を確認してください．:

```
x = Variable()
y = Variable()
p = Problem()
p += 2*x + 3*y
p += x + 2*y == 15
p += x >= 0
p += y >= 0
p.solve(silent=True)
#print(p.result)
Printf(' 関数の数          {}', p.result.nfunc)
Printf(' 内点法の反復回数 {}', p.result.iters)
Printf(' 関数評価回数      {}', p.result.fevals)
Printf(' 目的関数値        {}', p.result.optValue)
Printf(' 収束判定条件      {}', p.result.tolerance)
Printf(' 最適性条件残差    {}', p.result.residual)
Printf(' 所要計算時間      {}', p.result.elapseTime)
Printf(' 終了時ステータス {}', p.result.errorCode)
```

出力:

```
関数の数          4
内点法の反復回数 6
関数評価回数      9
目的関数値        22.500000000376254
収束判定条件      1e-08
最適性条件残差    3.762538162000488e-09
所要計算時間      0.004999995231628418
終了時ステータス 0
```

3.5.4 Fprintf 関数

Fprintf 関数は，標準出力ではなくファイルに対して出力をするための関数です．出力先が違うという点以外は，Printf 関数とほぼ同等の機能を有しています．Fprintf 関数の書式は以下のように定められています．出力先ファイルを指定するための第 1 引数以外は，Printf 関数と同様の書式です．

次の例では，変数の現在値を出力させています．出力ファイルとして，output.txt を指定しています．:

```
i = Element(value=[1, 2])
x = Variable(index=i, init={1: 10, 2: 20})
fp = open('output.txt', 'w')  # ファイルを開く
Fprintf(fp, '{}', x[i].val)
fp.close() # ファイルを閉じる
```

これに対する出力ファイル output.txt への出力は以下のようになります．:

```
10
20
```

Python の with 文を使うとより安全にファイルへの出力ができます．：

```
i = Element(value=[1, 2])
x = Variable(index=i, init={1: 10, 2: 20})
with open('output.txt', 'w') as fp:
    Fprintf(fp, '{}', x[i].val)
```

ファイルに上書きではなく，追加をしたい場合はファイルを開く際の引数を 'a' とする必要があります．：

```
with open('output.txt', 'a') as fp:
```

3.6 求解オプション

求解オプションは，求解時の動作をより細かく制御するためのものです．求解オプションを利用することにより，アルゴリズムの選択や，終了条件の調整などを行う事ができます．標準出力や解ファイルの制御については **問題クラス** *Problem* をご確認ください．求解オプションを設定する方法には問題クラスに対して options 属性を用います．選択型求解オプションについては求解オプション定数 Options を使用できます．

次の例ではアルゴリズムとして単体法 simplex を指定しています．

```
p = Problem()
p.options.method = 'simplex'
p.options.method = Options.Method.SIMPLEX
```

次の例では，残差停止条件を 10^{-12} に設定しています．

```
p.options.kktEps = 1e-12
```

次の例では，計算時間の上限を 60 秒に設定しています．

```
p.options.maxTime = 60
```

次の例では，分枝限定法にヒューリスティックサーチ rins を導入しています．：

```
p.options.branchRins = 1
p.options.branchRins = Options.Branch.Rins.ON
```

求解オプション定数の詳細については help(Options) を使用するか *Options* をご確認ください．

設定した値を削除してデフォルト値に戻すには del 文を用います．：

```
del p.options.maxTime
```

次の例では，すべての求解オプションを削除してデフォルト値に戻しています．：

```
del p.options
```

デフォルト値は 5. 求解オプション設定をご確認ください．

求解オプションの一覧は以下で確認できます．：

```
print(dir(p.options))
```

求解オプションの現在値一覧は以下で確認できます．：

```
print(p.options)
```

求解オプションの詳細については help(p.options) を使用するか *ProblemOptions* をご確認ください．

3.7 実行不可能性要因検出機能

デフォルトでは，実行不可能性を検出する iisDetect と呼ばれる仕組みが自動的に起動され，実行不可能性の原因の探索を行います（制約充足問題ソルバ使用時以外）．ここでは，iisDetect 機能が起動した場合の結果を説明します．

以下のモデル記述に書かれた線形計画問題は，実行不可能（制約を満たす解なし）です．：

```
x = Variable(lb=0)
y = Variable(lb=0)
z = Variable()
problem = Problem()
problem += x + y + z
problem += x >= 2 * y     , 'cons1'  # IIS
problem += 1 + 2 * z >= x, 'cons2'  # IIS
problem += y >= 2 + z     , 'cons3'  # IIS
problem += x >= z,          'cons4'
problem.solve()
assert problem.status == NuoptStatus.INFEASIBLE
```

よく見るとモデルで IIS のマークが付いた制約式群のどの一つを除去しても実行不可能性は解消しますが，すべてを満たす x, y, z は存在しません．また，マークされていない最後の制約式は実行不可能性とは無関係で，除去する，しないにかかわらず，問題は実行不可能であることもわかります．

iisDetect 機能はこのように，実行不可能性の原因となっている行の組 (Irreducible Infeasible Set: IIS) を特定して出力します．一般に実行不可能な問題について IIS は複数存在しますが，このアルゴリズムは可能な限り小さなもの（含まれている行が少ない）ものを求めるようなヒューリスティクスが導入されています．

IIS オブジェクトは result.iis 属性で取得することができます．：

```
problem.solve(silent=True)
print(problem.result.iis)
```

この出力は次のようになります．：

```
0: cons3: violation=1.5
    y-z>=2
1: cons2
    2*z-x>=-1
2: cons1
    x-2*y>=0
```

cons1, cons2, cons3, cons4 の制約のうち，cons1, cons2, cons3 が互いに矛盾する制約式の最小の組 (のひとつ) であることが分かりました．また，実行不可能ではあるものの，変数には何らかの値が設定されており，実際に制約違反を起こしている制約は cons3 で，違反量は 1.5 であることも分かります．

更に `problem.result.iis[0]` として詳細情報にアクセスすることもできます．詳細な属性は *IIS* をご確認ください．

モデルに非線形の式が含まれていた場合，IIS の正確な検出はできません．その場合には，ヘッダー部には IIS の検出が非線形性のために失敗したというメッセージが現れ，非線形な制約がいくつあるかを示します．

3.7.1 solfile オプション

solfile オプションは IIS を含めた解ファイルの出力を制御します．解ファイルはデフォルトでは出力されません．解ファイルは Problem オブジェクト生成時，または solve メソッド呼び出し時に solfile オプションで制御することができます．：

```
problem.solve(solfile=True)  # {problem.name}.sol が出力される
```

ファイル名を指定することもできます．：

```
problem.solve(solfile='myproblem')  # myproblem.sol が出力される
```

解ファイルの実行不可能性要因出力部は次のような出力になります．：

```
%%
%%    IIS
%%
--------------------------------------------------------------------------
#1      cons1         :   1*x-2*y
                                                >=          0   (         0)
--------------------------------------------------------------------------
#2      cons2         :   -1*x+2*z
                                                >=         -1   (        -1)
--------------------------------------------------------------------------
#3      cons3    INFS :   1*y-1*z
                                                >=          2   (       0.5)
--------------------------------------------------------------------------
```

iis オプションで iisDetect 機能をオフにしている場合は実行不可能性要因出力部は出力されません.

解ファイルの見方については 解ファイル をご確認ください.

3.7.2 iis オプション

iis オプションは iisDetect 機能のオン/オフを制御します. iisDetect 機能はデフォルトではオンになっています. :

```
problem.solve(silent=True)
print(problem.result.iis)
```

この出力は次のようになります. :

```
0: cons3: violation=1.5
    y-z>=2
1: cons2
    2*z-x>=-1
2: cons1
    x-2*y>=0
```

実行可能な場合や iisDetect 機能をオフにした場合など IIS が取得できない場合は空の IIS オブジェクトが設定されます. :

```
problem.solve(silent=True, iis=False)  # iisDetect 機能をオフにする
print(problem.result.iis)  # no IIS
```

iisDetect 機能をオフにすると実行不可能性要因検出が行われなくなるため, 実行不可能時の高速化が期待されます.

3.8 その他の機能

3.8.1 インポートについて

PySIMPLE でモデリングを行う前には使用するオブジェクトをインポートして使える状態にしておく必要があります. インポートには使用するオブジェクトだけを個別に行う方法と, すべてを一括して行う方法があります.

前者の方法では以下のように使用するオブジェクトをカンマで区切って並べます. :

```
from pysimple import Problem, Variable
```

後者の方法では * を使用することで PySIMPLE で使用可能なすべてのオブジェクトを利用することができるようになります. :

```
from pysimple import *
```

いずれの場合も Python の名前空間に存在する同名のオブジェクトを上書きすることで予期せぬ動作を起こす可能性があることに注意してください.

利用可能な PySIMPLE オブジェクトの一覧は以下のように確認することができます. :

```
$ python -i
Python 3.13.2 (tags/v3.13.2:4f8bb39, Feb  4 2025, 15:23:48) [MSC v.1942 64 bit␣
↪(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from pysimple import *
pysimple 1.6.0 (2025-02-17 16:12:35 +0900 c93cf43)
Copyright (C) 2019 NTT DATA Mathematical Systems Inc. All Rights Reserved.
>>> dir()
['Acos', 'Acosh', 'Asin', ..]
```

PySIMPLE をパッケージのまま利用することも可能です. :

```
>>> import pysimple
pysimple 1.6.0 (2025-02-17 16:12:35 +0900 c93cf43)
Copyright (C) 2019 NTT DATA Mathematical Systems Inc. All Rights Reserved.
>>> i = pysimple.Element(value=[1, 2], name='i')
>>> x = pysimple.Variable(index=i, name='x')
```

エイリアスを用いる場合は ps を推奨します. :

```
>>> import pysimple as ps
pysimple 1.6.0 (2025-02-17 16:12:35 +0900 c93cf43)
Copyright (C) 2019 NTT DATA Mathematical Systems Inc. All Rights Reserved.
>>> i = ps.Element(value=[1, 2], name='i')
>>> x = ps.Variable(index=i, name='x')
```

3.8.2 name 属性について

PySIMPLE では, 宣言時に name 属性を与えない **集合クラス** *Set*, **添字クラス** *Element*, **定数クラス** *Parameter*, **変数クラス** *Variable* の name 属性を自動判別します. 判別に失敗した場合クラス名となります. :

```
i = Element(value=[1,2])
x = Variable(index=i)
print(x)
print(Variable(index=i))
```

この出力は次のようになります. :

```
x:
x[1]
x[2]
Variable:
Variable[1]
Variable[2]
```

また，インタプリタでは必ず判別に失敗しますので name 属性を与えた方が良いでしょう．[1]

```
>>> i = Element(value=[1,2])
>>> x = Variable(index=i)
>>> x
Variable:
Variable[1]
Variable[2]
>>> y = Variable(index=i, name='y')
>>> y
y:
y[1]
y[2]
```

式 クラス, **制約式クラス** *Constraint* などの name 属性は構成要素から自動で与えられます．明示的に与える場合は name 属性を変更します．:

```
i = Element(value=[1, 2])
a = Parameter(index=i, value=3)
x = Variable(index=i)
ax = a[i] + x[i]
print(ax)
ax.name = 'ax'
print(ax)
```

出力:

```
(a[i]+x[i]):
x[1]+3
x[2]+3
ax:
x[1]+3
x[2]+3
```

[1] Python 3.13 からのインタプリタでは判別できるようになりました．

3.8.3 index 属性の便利な使い方

添字付きオブジェクトである Parameter, Variable, Expression, Constraint などは index 属性を持っています．この index 属性を活用すると，添字の次元を抽象化した記述をすることができます．以下では添字なしの変数 x と一次元の添字をもつ変数 xi に対して添字を抽象化して __getitem__ を利用しています．:

```
>>> i = Element(value=[1, 2], name='i')
>>> x = Variable(name='x')
>>> xi = Variable(index=i, name='xi')
>>> x[x.index]    # x と同じ
x:
x
>>> xi[xi.index]  # xi[i] と同じ
xi:
xi[1]
xi[2]
```

変数に 1 を加えた式を作成したい場合，通常ではそれぞれ x+1, x[i]+1 と記述する必要がありますが，index 属性を利用することで， x[x.index]+1 などと，添字の有無や次元を意識せず記述することができるようになります．

__getitem__ に似たメソッドとしてバージョン 1.5.0 で追加された get メソッドがあります．get メソッドは引数に可変長位置引数をとるため index 属性を展開して渡す必要があります．:

```
>>> x.get(*x.index)    # x.get() と同じ
x.get():
x.get()
>>> xi.get(*xi.index)  # xi.get(i) と同じ
xi.get(i):
xi.get(i)[1]
xi.get(i)[2]
```

同様に __setitem__ でも index 属性を利用することができます．特に添字を持たない変数 x では x=1 と記述することはできないため，index 属性を利用する明確なメリットがあります．:

```
>>> x[x.index] = 1    # x = 1 とは書けないので注意
>>> x.val
x.val=1
>>> xi[xi.index] = 1  # xi[i] = 1 と同じ
>>> xi.val
xi[1].val=1
xi[2].val=1
```

最後に宣言時の index 引数にも利用できることを記しておきましょう．ある変数の類似物を作成するときなど，添字を気にしなくて済むため便利な場合があります．:

```
>>> xval = Parameter(index=x.index, value=x.val, name='xval')
>>> xval
xval=1
>>> xival = Parameter(index=xi.index, value=xi.val, name='xival')
>>> xival
xival[1]=1
xival[2]=1
```

常に添字を抽象化する必要はありませんが，添字の次元が異なる変数を一括して処理したい場合など，有効な場面もあるでしょう．

3.8.4 pandas を用いた csv データの読み込み

本項では，データ解析分野で表操作によく使われる pandas モジュールを利用して，csv 形式で保持しているデータを PySIMPLE オブジェクトとして読み込む方法を紹介します．

添字が 1 つのデータ value, size を考えます．以下では knapsack.csv があるとします．：

```
i,value,size
缶コーヒー,120,10
水入りペットボトル,130,12
バナナ,80,7
りんご,100,9
おにぎり,250,21
パン,185,16
```

これを pandas を用いて読み込み，Parameter にするには次のようにします．：

```
import pandas as pd
# 1D, 1 index, 複数データ
data = pd.read_csv('knapsack.csv', index_col='i')  # この index_col は明示しなくてもよい
i = Element(value=data['value'].keys())  # 品物
value = Parameter(index=i, value=data['value'].to_dict())
size = Parameter(index=i, value=data['size'].to_dict())
```

添字が 2 つの場合も同様です．以下では assign.csv があるとします．：

```
i,j,gain,cost
1,A,31,13
1,B,12,60
1,C,19,76
2,A,76,32
2,B,75,120
2,C,15,6
```

これを読み込むには次のようにします．：

```
# 1D, 2 index, 複数データ
data = pd.read_csv('assign.csv', index_col=['i', 'j'])
ij = Element(value=data['gain'].keys())
gain = Parameter(index=ij, value=data['gain'].to_dict())  # 割当利益
cost = Parameter(index=ij, value=data['cost'].to_dict())  # 割当コスト
```

PySIMPLE で添字が複数次元のデータを与えるには key がタプルの辞書に変換する必要がありますが，これは DataFrame の MultiIndex を使うことで簡単に実現できます．read_csv 関数で読み込む場合，index_col キーワードで MultiIndex を指定することができます．

表形式の場合は少し加工が必要です．以下では product2.csv があるとします．:

```
i,A,B
0,60,40
1,10,60
```

これを読み込むには次のようにします．:

```
# 1D, 2 index
#data = pd.read_csv('product1.csv', index_col=['i', 'j'])

# 2D, 2 index
data = pd.read_csv('product2.csv')
data = data.melt(id_vars='i', var_name='j', value_name='product')
data = data.set_index(['i', 'j'])
ij = Element(value=data['product'].keys())
product = Parameter(index=ij, value=data['product'].to_dict())
```

表形式の場合，read_csv 関数で読み込んだ後，melt 関数で整形し，set_index 関数を用いることで MultiIndex を指定することができます．

表形式で添字が 3 つの場合も同様です．以下では demand2.csv があるとします．:

```
i,j,X,Y
0,A,300,200
0,B,180,150
1,A,240,180
1,B,150,120
```

これを読み込むには次のようにします．:

```
# 1D, 3 index
#data = pd.read_csv('demand1.csv', index_col=['i', 'j', 'k'])

# 2D, 3 index
data = pd.read_csv('demand2.csv')
data = data.melt(id_vars=['i', 'j'], var_name='k', value_name='demand')
```

(次のページに続く)

(前のページからの続き)

```
data = data.set_index(['i', 'j', 'k'])
ijk = Element(value=data['demand'].keys())
demand = Parameter(index=ijk, value=data['demand'].to_dict())
```

3.8.5 for 文を使う

for 文は要素を逐次処理するための制御フローです．PySIMPLE の集合，定数，変数，式，制約式などのオブジェクトはいずれも for 文で処理することができます．:

```
IJ = Set(value=[(1,3), (1,4), (2,3)])
for elm in IJ:
    print(elm)
```

この出力は次のようになります．:

```
(1, 3)
(1, 4)
(2, 3)
```

出力順は毎回同じになります．

添字では set 属性で対応する集合を用います．:

```
ij = Element(value=[(1,3), (1,4), (2,3)])
for elm in ij.set:
    print(elm)
```

次元に限らず各要素は必ずタプルになることに注意してください．:

```
I = Set(value=[1, 2])
for elm in I:
    print(elm)
```

出力:

```
(1,)
(2,)
```

一次元の要素をタプルとして扱いたくない場合，アンパックしながら取り出すと簡単に回避できます．:

```
I = Set(value=[1, 2])
for elm, in I:  # 'elm,' とカンマがついている
    print(elm)
```

出力:

```
1
2
```

定数は辞書と同じように扱われます．すなわちキーが取り出されます．:

```
i = Element(value=[1, 2])
a = Parmeter(index=i, value={1: 10, 2: 20})
for elm in a:
   print(elm)
```

出力:

```
(1,)
(2,)
```

値を取り出すには `values()`，キーと値を同時に取り出すには `items()` メソッドを使います．:

```
for key, value in a.items():
   print(key, value)
```

出力:

```
(1,), 10
(2,), 20
```

変数の値を取り出すには `.val` などの属性に続けて同様の操作ができます．:

```
i = Element(value=[1, 2])
x = Variable(index=i, init={1: 10, 2: 20})
for key, value in x.val.items():
    print(key, value)
```

3.8.6 Python オブジェクトへの変換

次の例は集合を Python のリストに変換しています．:

```
I = Set(value=[1,2,3])
print(list(I))
```

出力:

```
[(1,), (2,), (3,)]
```

次元に限らず各要素は必ずタプルになることに注意してください．

次の例は定数を Python の辞書に変換しています．:

```
i = Element(value=[1, 2])
a = Parameter(index=i)
a[i] = i
print(dict(a))
```

出力:

```
{(1,): 1, (2,): 2}
```

次の例は変数の現在値を Python のリストに変換しています．:

```
x = Variable(index=i)
x[i] = i
print(list(x.val.values()))
```

出力:

```
[1, 2]
```

次の例は変数の現在値を Python の辞書に変換しています．:

```
x = Variable(index=i)
x[i] = i
print(dict(x.val.items()))
```

出力:

```
{(1,): 1, (2,): 2}
```

次の例は定数を Python の整数型に変換しています．:

```
a = Parameter(value=3)
print(int(a))
```

出力:

```
3
```

次の例は変数の値を Python の浮動小数点型に変換しています．:

```
x = Variable(init=3)
print(float(x.val))
```

出力:

```
3.0
```

添字を含むオブジェクトは整数型や浮動小数点型へ変換できません．次の記述は誤りです．:

```
i = Element(value=[1, 2])
a = Parameter(index=i)
a[i] = i
print(int(a))
```

3.8.7 pandas を用いた結果表の出力

割当問題などでは最適化の結果を表形式で見たいケースが多くあります．本項では，データ解析分野で表操作によく使われる pandas モジュールの DataFrame を用いて結果を加工します．

まずは簡単な割当問題を見てみましょう．:

```
i = Element(value=range(4))  # 仕事
j = Element(value='ABCD')    # 人
x = BinaryVariable(index=(i,j))
p = Problem()
p += Sum(x[i,j], i) == 2, '各人は 2 つの仕事をする'
p += Sum(x[i,j], j) == 2, '各仕事は 2 人で分担する'
p.solve(silent=True)
```

一般に最適化の出力は整然データ (tidy data) の形式になります．整然データを表形式に加工するには pivot 関数を使うことでできます．:

```
import pandas as pd
output = pd.DataFrame((x[i,j].val>0).set, columns=['i','j'])
output['val'] = 1
print(output.pivot(*output.columns).fillna(0).astype(int))
```

出力:

```
j  A  B  C  D
i
0  0  0  1  1
1  1  0  1  0
2  0  1  0  1
3  1  1  0  0
```

データ加工の二行目ではまず，`x[i,j].val>0` で正の値が入っている部分からなる (二次元の) 添字を抽出しています．添字自体はイテレートできないので，*set* 属性により長さ 2 のタプル列を DataFrame に渡します．三行目では値列を追加し，四行目でピボットテーブルに加工します．pivot 関数は index, columns, values 引数を取りますが，それぞれが output の i 列，j 列，val 列に対応するため，`output.columns` を渡すことで簡略化しています．次のように，最初から三つ組を DataFrame に与えることもできます．:

```
val = [(*ij, v) for ij, v in x.val.items()]
output = pd.DataFrame(val, columns=['i','j', 'val'])
output.pivot(*output.columns).astype(int)
```

次は，別の例としてシフト表を作ってみましょう．5 人の一週間のシフトとして日勤 (Day)・夜勤 (Ngt)・休み (Off) を決めます．:

```
m = Element(value='ABCDE')      # Man
d = Element(value=range(1,8))   # Day
s = Element(value=['Day','Ngt','Off'])  # Shift
prb = Problem()
x = BinaryVariable(index=(m,d,s))
prb += Sum(x[m,d,s], s) == 1, '一日に勤務は一つ'
prb += Sum(x[m,d,'Day'], m) == 2, '日勤は二人ちょうど'
prb += Sum(x[m,d,'Ngt'], m) >= 1, '夜勤は一人以上'
prb += Sum(x[m,d,'Off'], d) >= 1, '一回以上は休み'
prb += Sum(x[m,d,'Ngt'], d) <= 2, '夜勤は二回まで'
d1 = d != 7
prb += x[m,d1+1,'Off'] >= x[m,d1,'Ngt'], '夜勤の次の日は代休'
prb.solve(silent=True)
```

今回は，DataFrame への値の与え方が少し異なり，添字自体が三次元です．:

```
output = pd.DataFrame((x[m,d,s].val>0).set, columns=['Man','Day','Shift'])
piv = output.pivot(*output.columns)
print(piv)
```

出力:

```
Day    1    2    3    4    5    6    7
Man
A    Day  Off  Day  Ngt  Off  Day  Ngt
B    Off  Day  Off  Day  Day  Off  Ngt
C    Off  Ngt  Off  Day  Ngt  Off  Day
D    Ngt  Off  Ngt  Off  Off  Day  Day
E    Day  Day  Day  Off  Day  Ngt  Off
```

制約を満たしているか確認のため集計列を集計行を付け加えてみましょう．:

```
piv.loc['DayCount'] = (piv == 'Day').sum(axis=0)
piv.loc['NgtCount'] = (piv == 'Ngt').sum(axis=0)
piv = piv.assign(OffCount=(piv == 'Off').sum(axis=1),
                 NgtCount=(piv == 'Ngt').sum(axis=1))
print(piv)
```

出力:

```
Day         1    2    3    4    5    6    7  OffCount  NgtCount
Man
A         Day  Off  Day  Ngt  Off  Day  Ngt         2         2
B         Off  Day  Off  Day  Day  Off  Ngt         3         1
```

(次のページに続く)

(前のページからの続き)

```
C          Off  Ngt  Off  Day  Ngt  Off  Day          3          2
D          Ngt  Off  Ngt  Off  Off  Day  Day          3          2
E          Day  Day  Day  Off  Day  Ngt  Off          2          1
DayCount     2    2    2    2    2    2    2          0          0
NgtCount     1    1    1    1    1    1    2          0          0
```

3.8.8 PySIMPLE オブジェクトのシリアライズ

Python には，Python オブジェクトをシリアライズ (直列化) できる pickle という機能が標準で用意されています．PySIMPLE では，pickle と同様に PySIMPLE オブジェクトをシリアライズできる Serialize クラスがあります．

以下では，添字付き変数をシリアライズし，ファイルに dump，復元をしています．：

```
i = Element(value=[1, 2, 3])
x = Variable(index=i)
x[i] = i*10
with open('dump.pkl', 'wb') as f:
    Serialize.dump(x, f)

with open('dump.pkl', 'rb') as g:
    x_ = Serialize.load(g)
print(x_.val)
```

以下では，Problem オブジェクトをファイルに dump，復元をした後，求解しています．この使用法は求解直前から再開できるため，求解前までに時間を要したり，手間がかかる場合に有効です．：

```
i = Element(value=[1, 2])
x = Variable(index=i, lb=0, ub=5)
problem = Problem()
problem += 6*x[1] +   x[2] >= 12
problem += 4*x[1] + 6*x[2] >= 24
problem += 180*x[1] + 160*x[2]

# Problem をファイルに dump
with open('dump.pkl', 'wb') as f:
   Serialize.dump(problem, f)

# 読み込み
with open('dump.pkl', 'rb') as g:
   problem_ = Serialize.load(g)

x_ = problem_.variables['x']
```

(次のページに続く)

(前のページからの続き)

```
problem_.solve(silent=True)
print(x_.val)
```

以下では，求解後に Problem に登録した全ての変数をファイルに dump，復元後，元の変数に値を設定しています．この使用法は求解時の値を簡単に復元できるため，実行可能な初期値を設定する，実行不可能時に以前の値に戻す，といったケースが考えられます．:

```
i = Element(value=[1, 2])
x = Variable(index=i, lb=0, ub=5)
problem = Problem()
problem += 6*x[1] +   x[2] >= 12
problem += 4*x[1] + 6*x[2] >= 24
problem += 180*x[1] + 160*x[2]
problem.solve(silent=True)
print(x.val)

# Problem に登録した全ての変数をファイルに dump
with open('dump.pkl', 'wb') as f:
   Serialize.dump(list(problem.variables.values()), f)

# 変数の値を変える操作
x[i] = 0
print(x.val)  # 0

# 読み込み
with open('dump.pkl', 'rb') as g:
   variables = Serialize.load(g)

# 元の変数に復元した値を設定
for var1, var2 in zip(problem.variables.values(), variables):
   # 今回の場合 x[i] = variables[0][i].val と等価
   var1[var1.index] = var2[var1.index].val
print(x.val)
```

また，Serialize クラスは Python の pickle モジュールのラッパーであるため，Python オブジェクトもシリアライズできます．

メソッド一覧は *Serialize* をご確認ください．

3.8.9 境界条件やフロー保存則の記述テクニック

PySIMPLE で扱う添字付きオブジェクトは範囲外の添字にアクセスすると通常 KeyError 例外を投げます．：

```
>>> i = Element(value=[1, 2, 3], name='i')
>>> z = BinaryVariable(index=i, name='z')
>>> z[i+1]
KeyError: 'z[(i+1)[i]][4]'
```

上記の場合，i=3 のときに z[4] にアクセスしているため，KeyError となります．V26 で新たに導入された get メソッドでは Python の辞書における get メソッドのように範囲外にアクセスしても例外とならず，PySIMPLE においては，0 となります．

モデルを記述する上で頻出である境界値で分岐する制約式を get メソッドなし・ありで記述してみましょう．get メソッドを用いることで制約式を二つに分割することなく，一括して記述することができます．：

```
>>> # without get
>>> i1 = i!=1  # i1 in (2, 3)
>>> z[1] >= 1
(z[1]>=1):
z[1]>=1
>>> z[i1] - z[i1-1] >= 1
((z[(i!=1)]-z[((i!=1)-1)[(i!=1)]][(i!=1)])[(i!=1)]>=1):
-z[1]+z[2]>=1
-z[2]+z[3]>=1
>>> # with get
>>> z[i] - z.get(i-1) >= 1
((z[i]-z.get((i-1)[i])[i])[i]>=1):
z[1]>=1
-z[1]+z[2]>=1
-z[2]+z[3]>=1
```

今度はフローネットワークの保存則を get メソッドなし・ありで記述してみましょう．入力のみのノード 1，出力のみのノード 6 の扱いに注意すると，get メソッドなしでは制約式を三つに分割する必要があるのに対し，get メソッドありでは一括して記述することができます．：

```
>>> IJ = Set(value=[(1,2),(1,4),(2,3),(3,6),(4,5),(5,2),(5,6)], name='IJ')
>>> ij = Element(set=IJ, name='ij')
>>> x = Variable(index=ij, name='x')
>>> # without get
>>> m = Element(set=IJ(0)&IJ(1), name='m')  # m in (2, 3, 4, 5)
>>> Sum(x[ij], ij(0))[m] == Sum(x[ij], ij(1))[m]  # 入出力あり
(Sum(x[ij], ij(0))[m]==Sum(x[ij], ij(1))[m]):
x[1,2]-x[2,3]+x[5,2]==0
x[1,4]-x[4,5]==0
x[2,3]-x[3,6]==0
x[4,5]-x[5,2]-x[5,6]==0
```

(次のページに続く)

(前のページからの続き)

```
>>> Sum(x[ij], ij(1))[ij(0)>IJ(1)] == 0  # ij(0)>IJ(1) in (1,)
(Sum(x[ij], ij(1))[(ij(0)>IJ(1))]==0):
x[1,2]+x[1,4]==0
>>> Sum(x[ij], ij(0))[ij(1)>IJ(0)] == 0  # ij(1)>IJ(0) in (6,)
(Sum(x[ij], ij(0))[(ij(1)>IJ(0))]==0):
x[3,6]+x[5,6]==0
>>> # with get
>>> k = Element(set=IJ(0)|IJ(1), name='k')  # k in (1, ..., 6)
>>> Sum(x[ij], ij(0)).get(k) == Sum(x[ij], ij(1)).get(k)
(Sum(x[ij], ij(0)).get(k)[k]==Sum(x[ij], ij(1)).get(k)[k]):
-x[1,2]-x[1,4]==0
x[1,2]-x[2,3]+x[5,2]==0
x[2,3]-x[3,6]==0
x[1,4]-x[4,5]==0
x[4,5]-x[5,2]-x[5,6]==0
x[3,6]+x[5,6]==0
```

get メソッドを用いて一括して記述することで，見た目が簡潔になるだけでなく，その後に式の値を取得したり制約式を削除したりするなど，制御も簡単になります．get メソッドは C++SIMPLE には存在しない PySIMPLE オリジナルの文法です．非常に強力な記述力を持ちますが，一方で全く異なる式を一括して記述することには向いていません．

3.8.10 多次元の添字のテクニック

多次元の添字は便利な面もありますが，次元が多くなると少し見づらくなることもあります．以下のモデルを見てみましょう．:

```
>>> ijk = Element(value=[(1,3,5), (1,3,6), (1,4,5), (2,3,5)], name='ijk')
>>> x = Variable(index=ijk(0,1), name='x')
>>> a = Parameter(index=ijk(1), value=1, name='a')
>>> Sum(x[ijk(0,1)], ijk(0,2)) >= a[ijk(1)]  # 通常の書き方
(Sum(x[ijk(0,1)], ijk(0,2))[ijk(1)]>=a[ijk(1)]):
2*x[1,3]+x[2,3]>=1
x[1,4]>=1
```

こんなとき PySIMPLE ではエイリアスを使うことで記述を簡潔にすることができます．:

```
>>> i, j, k = ijk(0), ijk(1), ijk(2)  # エイリアスを使った書き方
>>> Sum(x[i,j], (i,k)) >= a[j]
(Sum(x[ijk(0),ijk(1)], (ijk(0),ijk(2)))[ijk(1)]>=a[ijk(1)]):
2*x[1,3]+x[2,3]>=1
x[1,4]>=1
```

多次元の添字にも拘らず，その疎性を気にすることなくモデルを記述することができています．また，エイリアスの定義部分は以下のようにして汎用化することも可能です．:

```
>>> i, j, k = map(ijk, range(ijk.set.dim))  # 汎用化
>>> Sum(x[i,j], (i,k)) >= a[j]
(Sum(x[ijk(0),ijk(1)], (ijk(0),ijk(2)))[ijk(1)]>=a[ijk(1)]):
2*x[1,3]+x[2,3]>=1
x[1,4]>=1
```

3.8.11 次元を落としたオブジェクト作成

選択関数 *Selection* に伴う，次元を落としたオブジェクト作成の方法について紹介します．以下では変数 `z[i,j]` は Selection 制約のため，各 `i` について `z[i,j].val=1` となる `j` が存在します．:

```
>>> i = Element(value=[1, 2, 3], name='i')
>>> j = Element(value=['X', 'Y'], name='j')
>>> z = BinaryVariable(index=(i,j), name='x')
>>> p = Problem()
>>> p += Selection(z[i,j], j)
>>> p.solve(silent=True)
<NuoptStatus.OPTIMAL: 1>
>>> ij = z[i,j].val>0
>>> z[ij].val
z[1,'X'].val=1
z[2,'Y'].val=1
z[3,'Y'].val=1
```

これを `i` をキーとしたオブジェクトとして扱いたい場合にはどうすればよいでしょうか．以下では `z[i,j].val>0` を満たす `(i,j)` の組合せ `ij` を用いることにより，`i` をキー，対応する `j` を値とするパラメーター `a` を作成しています．:

```
>>> a = Parameter(index=i, name='a')
>>> a[ij(0)] = ij(1)
>>> a
a[1]='X'
a[2]='Y'
a[3]='Y'
```

更に `a` は一度に `Parameter(index=i, value=dict(ij.set))` と記述することも可能です．

Selection で和をとった後に残る添字 (上記では `i`) が多次元の場合でも同様の方法で扱うことができます．では，和をとる添字 (上記では `j`) が多次元だった場合はどうすればよいでしょうか．以下では `z[i,j,k]` を `(j,k)` で Selection をとっているため，各 `i` について `z[i,j,k].val=1` となる `(j,k)` の組が存在します．:

```
>>> i = Element(value=[1, 2, 3], name='i')
>>> j = Element(value=['A', 'B'], name='j')
>>> k = Element(value=['X', 'Y'], name='k')
>>> z = BinaryVariable(index=(i,j,k), name='z')
```

(次のページに続く)

(前のページからの続き)

```
>>> p = Problem()
>>> p += Selection(z[i,j,k], (j,k))
>>> p.solve(silent=True)
<NuoptStatus.OPTIMAL: 1>
>>> ijk = z[i,j,k].val>0
>>> z[ijk].val
z[1,'A','Y'].val=1
z[2,'B','X'].val=1
z[3,'B','Y'].val=1
```

Parameter は一次元の値しかとることができないため，このような場合， j 部分を担当するパラメーター aj と， k 部分を担当するパラメーター ak に分割することで，i をキーとしたオブジェクトとして扱うことができるようになります．このようにすることで (j,k) の組合せが欲しい場合には aj[i] と ak[i] と記述できます．:

```
>>> aj = Parameter(index=i, name='aj')
>>> aj[ijk(0)] = ijk(1)
>>> aj
aj[1]='A'
aj[2]='B'
aj[3]='B'
>>> ak = Parameter(index=i, name='ak')
>>> ak[ijk(0)] = ijk(2)
>>> ak
ak[1]='Y'
ak[2]='X'
ak[3]='Y'
>>> Printf('{}: {}, {}', i, aj[i], ak[i])
1: A, Y
2: B, X
3: B, Y
```

更に aj, ak はそれぞれ Parameter(index=i, value=dict(ijk(0,1).set)), Parameter(index=i, value=dict(ijk(0,2).set)) と記述することも可能です．

3.8.12 LP の値を丸めて MILP の近似解を得るテクニック

混合整数線形計画問題 (MILP) は線形計画問題 (LP) に比べ難しく，実行可能解や最適解が求まらないこともあります．一方で MILP の整数変数を連続変数に緩和した問題 (連続緩和問題) は線形計画問題のため，ずっと簡単になります．本項では連続緩和問題の丸め値を元問題の近似値として与えることで近似解を求める手法を紹介します．:

```
def problem(type):
    x = Variable(type=type, lb=0, ub=5)
```

(次のページに続く)

(前のページからの続き)

```
    y = Variable(lb=0, ub=5)
    p = Problem()
    p += 180*x + 160*y
    p += 6*x +   y >= 12
    p += 4*x + 6*y >= 24
    return p

rp = problem(type=float)  # 連続緩和問題 (rp は relaxed problem の頭文字)
rp.solve()                # 連続緩和問題を解く (LP なので簡単)
rx = rp.variables['x']
rx[rx.index] = round(rx.val)  # 整数値に丸めた値を設定
rx.fix()                  # x の値を固定
rp.solve()
print(rx.val, rp.objective.val)
```

problem は type=int とすれば元問題の MILP を，type=float とすればその連続緩和問題を返す関数です．fix メソッドは変数の上下限を現在の値に固定します．すなわち今回の場合は「rp += rx == rx.val」という制約と同じ意味ですが，fix は制約ではなく上下限扱いとなります．また，後述の unfix で固定状態を細かく制御できる点でも異なります．

連続緩和問題で得られた x の値 (実数値) を整数に丸めて固定し，再求解することで MILP を解くことなく整数性を満たした求解を行っています．ただし，この解はあくまで近似解であり，実行可能解である保証もありませんが，元問題が難しい場合は検討の余地があるでしょう．また，丸め値を設定する箇所は「rx = ...」と記述できないことに注意しましょう．このように記述してしまうと rx の参照先が上書きされてしまいます．なお，「rx[rx.index] = ...」という記述は添字の有無に拘わらず利用できます．

上記では元問題の整数変数が x ひとつでしたが，以下のように汎用的に記述することで，モデルに含まれるすべての整数変数に丸め値を固定することができます．:

```
p = problem(type=int)     # 元問題 (MILP)
rp = problem(type=float)  # 連続緩和問題 (LP)
rp.solve()
for rvar, var in zip(rp.variables.values(), p.variables.values()):
    if var.type is not float:  # 元問題で整数変数の変数のみ
        rvar[rvar.index] = round(rvar[rvar.index].val)
        rvar.fix()
rp.solve()
print(rp.objective.val)
```

以下に今回用いた問題の最適解をまとめておきます．(IP は参考)

class	x.type	y.type	x.val	y.val	目的関数値
LP	float	float	1.5	3.0	750.0
MILP	int	float	2.0	2.7	786.7
IP	int	int	2.0	3.0	840.0

連続緩和問題で得られた x の値 1.5 を整数に丸めた値 2.0 は元問題における最適解と一致していますが，偶然であることに注意しましょう．実際，x の値を 1.0 に丸めた場合は実行不可能となります．実行不可能な場合に一部の整数変数の固定を解除するといった対応を考えてみましょう．

次の例は，一度 fix メソッドで固定した変数の値の一部を unfix メソッドを用いて解除しています．このような部分的な解除は制約式では行うことができません．:

```
i = Element(value=[1, 2, 3])
z = IntegerVariable(index=i, lb=10, init=20)
p = Problem()
p += Sum(z[i])
z.fix()       # p += z[i] == z[i].val, 'cons' と意味は同じ
p.solve()
print(z.val)  # {1: 20, 2: 20, 3: 20}
z[2].unfix()  # del p['cons'][2] はできない
p.solve()
print(z.val)  # {1: 20, 2: 10, 3: 20}
```

3.8.13 高速化 TIPS

本項では，PySIMPLE の性能を引き出すことのできる高速化方法を 4 つ紹介します．

Set/Element の value には range を与える

Set の value キーワードには iterable オブジェクト全般を渡すことができますが，range の場合は専用の高速化処理が行われ，下記の例だと，10 倍程度の速度差が発生します．Element にも value キーワードがありますが同様です．添字が等差数列の場合はなるべく range オブジェクトを渡すようにしましょう．:

```
I = Set(value=list(range(1000000)))  # 遅い
I = Set(value=range(1000000))        # 速い
```

set が同じ添字は使いまわす

Set や Element に value が与えられた場合は内部でデータチェックが行われます．前者ではデータチェックが 2 回，後者では 1 回であるため，後者の方が速くなります．:

```
# 遅い
i1 = Element(value=range(1000000))
i2 = Element(value=range(1000000))
# 速い
I = Set(value=range(1000000))
i1 = Element(set=I)
i2 = Element(set=I)
```

後者は次のように記述することもできます．:

```
i1 = Element(value=range(1000000))
i2 = Element(set=i1.set)
```

制約より lb/ub を使う

PySIMPLE では変数の宣言時に lower/upper bound を与えることができます．bound に定数を渡した場合は，与えない場合と同じ速度で初期化が行われるため，後者では制約式の分，高速になります．:

```
i = Element(value=range(1000000))
p = Problem()
# 遅い
x = Variable(index=i)
p += x[i] >= 1
# 速い
x = Variable(index=i, lb=1)
```

データは宣言時に与える

PySIMPLE では Parameter の値や Variable の初期値を宣言時に与えることも，宣言後に変更することもできます．しかし，Parameter/Variable への代入は添字つき処理を行うことを目的として実装されているため，通常の代入に比べると遅くなってしまいます．そのため，個別に値を指定したい場合は辞書を作成しておき，宣言時に与える方が効率的です．:

```
# 遅い
a = Parameter(index=i)
for idx in ...:
    a[idx] = ...

# 速い
data = {}
for idx in ...:
    data[idx] = ...
a = Parameter(index=i, value=data)
```

一方，次のような添字付き代入の場合は問題ありません．:

```
a = Parameter(index=i)
a[i] = i
```

3.8.14 実数緩和について

次の数理計画問題を考えてみましょう．

整数変数
$x_i, \quad i \in \{1, 2\}$
制約
$x_i \geq 0.1, \quad \forall i$
$2 \sum_i x_i \leq 3$
目的関数 (最大化)
$\sum_i x_i$

実はこの問題には整数解は存在しませんが，実数変数であれば実行可能解が存在します．次の例は数理計画問題を一度 整数計画問題として求解し，実行不可能だったら実数変数として解き直しています．:

```
def showresult(p, x):
    if p.isFeasible():
        print('feasible!')
        print(x.val)
    elif p.isInfeasible():
        print('infeasible!')
    else:
        print('some error!')

def problem(vartype):
    p = Problem(type=max, silent=True)
    i = Element(value=[1, 2])
    x = Variable(index=i, lb=0.1, type=vartype)
    p += 2*Sum(x[i]) <= 3
    p += Sum(x[i])
    p.solve()
    showresult(p, x)
    return p.isInfeasible()

is_infeasible = problem(int)
if is_infeasible:
    print('relax!')
    problem(float)
```

この出力は次のようになります．:

```
infeasible!
relax!
feasible!
x[1].val=0.7499999998309204
x[2].val=0.7499999998309204
```

上記の例ではほぼ同じ問題を再度定義しているため冗長と言えます．変数の type 属性を後から変更できる性質を利用すると，問題はそのままで実数緩和を行うことができます．:

```
p = Problem(type=max, silent=True)
i = Element(value=[1, 2])
x = IntegerVariable(index=i, lb=0.1)
p += 2*Sum(x[i]) <= 3
p += Sum(x[i])

p.solve()
showresult(p, x)
```

(次のページに続く)

(前のページからの続き)

```
if p.isInfeasible():
    print('relax!')
    x.type = float  # 実数変数に変更
    p.solve()
    showresult(p, x)
```

type 属性の変更には添字をつけることはできません．次の記述は誤りです．:

```
x[i].type = float
```

3.8.15 スパースなモデルの書き方

多次元の添字を使うことによる，疎なモデルの書き方をみていきます．疎なモデルとして扱うことにより，無駄な変数や制約式が減り，より大きな問題を解くことができるようになったり，高速化が期待できます．

まずは，簡単な輸送問題を例に考えてみましょう．

- 倉庫 d と顧客 c が複数ある
- 倉庫には取扱上限 upper が存在する
- 顧客には需要量 lower が存在する
- 輸送コスト cost がかかる
- 総輸送コストを少なくしたい

この問題を単純に PySIMPLE で実装すると次のようになります．:

```
d = Element(value=uppervalue.keys())  # 倉庫
c = Element(value=lowervalue.keys())  # 顧客

# 倉庫から顧客への輸送コスト
cost = Parameter(index=(d,c), value=costvalue)
upper = Parameter(index=d, value=uppervalue)  # 倉庫取扱量上限
lower = Parameter(index=c, value=lowervalue)  # 顧客需要量下限
z = Variable(index=(d,c), lb=0)  # 倉庫から顧客への輸送量

prb = Problem()
prb += Sum(z[d,c], c) <= upper[d], '倉庫取扱量上限'
prb += Sum(z[d,c], d) >= lower[c], '顧客需要量下限'
prb += Sum(cost[d,c]*z[d,c]), '総輸送コスト'  # 目的関数
prb.solve()
```

ここで輸送コストは単位あたりの輸送費とし，(倉庫，顧客) ごとに決まるとします．そのため入力データ costvalue は以下のようになります．:

```
costvalue = {('d1', 'c1'): 30,
             ('d1', 'c2'): 20,
             ('d1', 'c3'): 9999,
             ('d1', 'c4'): 9999,
             ('d2', 'c1'): 25,
              :
}
```

9999 としているのは実際には輸送が発生しない経路に対するダミーの値です．大きな値を入力しておくことで輸送が発生し辛くしています．

このように現実問題ではすべての輸送経路を考慮する必要はなく，実際に輸送する経路はごく一部であるケースも多いでしょう．このような場合，すべての経路を用意しておくことは変数・制約式の数を無駄に増やしてしまうため，大きな問題が解けなかったり，速度が遅くなってしまいます．

そこで必要最小限の経路のみを定義することにより効率的なモデルを記述することができます．PySIMPLE では倉庫から顧客への二次元の添字 dc を用いることで疎なモデルを表現することができます．:

```
# 倉庫と顧客の疎な添字
dc = Element(value=costvalue.keys())

# 倉庫から顧客への輸送コスト
cost = Parameter(index=dc, value=costvalue)
upper = Parameter(index=dc(0), value=uppervalue)  # 倉庫取扱量上限
lower = Parameter(index=dc(1), value=lowervalue)  # 顧客需要量下限
z = Variable(index=dc, lb=0)  # 倉庫から顧客への輸送量

prb = Problem()
prb += Sum(z[dc], dc(1)) <= upper[dc(0)], '倉庫取扱量上限'
prb += Sum(z[dc], dc(0)) >= lower[dc(1)], '顧客需要量下限'
prb += Sum(cost[dc]*z[dc]), '総輸送コスト'  # 目的関数
prb.solve()
```

添字 (d,c) の代わりに dc を用いることで経路のある部分だけを表現することができました．倉庫 d や顧客 c の一方のみを指定したい場合は dc(0)，dc(1) と記述します．次元は 0 始まりであることに注意しましょう．また，costavalue は必要部分だけ記述すればよいので次のようになります．:

```
costvalue = {('d1', 'c1'): 30,
             ('d1', 'c2'): 20,
             ('d2', 'c1'): 25,
              :
}
```

輸送のありえる経路が全ての組合せの 20% だった場合，変数の 8 割が削減されたことになります．

疎なモデルを用いて効率的なモデルを記述する方法は特にネットワーク系の問題で威力を発揮します．他にも例えばスケジューリング問題では考慮する必要のないマスをあらかじめ除いておくこともできます．

3.8.16 パラメータを変更した逐次求解

パラメータを変更した逐次求解を行いたい場合，C++SIMPLE では可変定数 VariableParameter を利用できます．通常，式に登場する定数は式を定義したときの値で評価されますが，可変定数はこれを後から変更することができる点で Parameter と異なります．PySIMPLE には C++SIMPLE の可変定数はありませんが，同じ意味の記述は可能です．Nuorium Optimizer C++SIMPLE マニュアルで紹介されている直線の傾き a を変化させながら円と直線の接点を求めるモデルを書いてみましょう．:

```
x = Variable()  # 円の x 座標
y = Variable()  # 円の y 座標
p = Problem(type=max)
p += (x - 1)**2 + (y + 0.5)**2 <= 0.25  # (1, -0.5) を中心とする円
for a in range(-5, 5):
    p += -a*x + y, 'obj'
    p.solve(silent=True)
    assert p.status == NuoptStatus.OPTIMAL
    print(f'{a=:2} {x.val:.3f} {y.val:.3f} {p.objective.val:.3f}')
```

PySIMPLE では通常，目的関数を一度しか設定できませんが，求解をはさむことで再度設定ができるようになります．このモデルの出力は次のようになります．:

```
a=-5 x.val=1.490 y.val=-0.402 obj.val=7.050
a=-4 x.val=1.485 y.val=-0.379 obj.val=5.562
a=-3 x.val=1.474 y.val=-0.342 obj.val=4.081
a=-2 x.val=1.447 y.val=-0.276 obj.val=2.618
a=-1 x.val=1.354 y.val=-0.146 obj.val=1.207
a= 0 x.val=1.000 y.val=-0.000 obj.val=-0.000
a= 1 x.val=0.646 y.val=-0.146 obj.val=-0.793
a= 2 x.val=0.553 y.val=-0.276 obj.val=-1.382
a= 3 x.val=0.526 y.val=-0.342 obj.val=-1.919
a= 4 x.val=0.515 y.val=-0.379 obj.val=-2.438
```

上記は目的関数を変更するモデルでしたが，制約式を変更するモデルも見てみましょう．以下は二次関数の最小値を定義域を変更しながら求めるモデルです．:

```
x = Variable()
p = Problem()
p += x**2 - 4*x + 8, 'obj'
for a in range(5):
    p += x >= a, 'cons'
    p.solve(silent=True)
    assert p.status == NuoptStatus.OPTIMAL
    del p['cons']  # del p[-1] でも可
    print(f'{a=} {x.val:.2f} {p.objective.val:.2f}')
```

PySIMPLE では += 演算子では制約式の上書きを行うことはできないため，求解後に del 文を用いて一度削除しています．一方で， `p.constraints['cons'] = x >= a` と記述すれば制約式の上書きを行うことが

可能です．この場合は del 文は不要となります．このモデルの出力は次のようになります．：

```
a=0 x.val=2.00 obj.val=4.00
a=1 x.val=2.00 obj.val=4.00
a=2 x.val=2.00 obj.val=4.00
a=3 x.val=3.00 obj.val=5.00
a=4 x.val=4.00 obj.val=8.00
```

これまでは目的関数・制約式が単純でしたが，複雑な場合には少し注意が必要です．例えば problem += 重い式 >= 可変定数 のような制約式の場合，可変定数に無関係である重い式がループの度に評価されてしまいます．これを避けるには heavyexp = 重い式 と重い式をループの外に出し， problem += heavyexp >= 可変定数 とすることで重い式の評価を一度に抑えることが可能となります．また，可変定数を用いた制約式が多いようであれば，問題自体を関数化してしまうのもよいでしょう．

3.8.17 多目的最適化

多目的最適化問題とは目的関数が複数存在する数理最適化問題のことです．複数の目的関数を同時に最適とする解 (完全最適解) が存在するとは限らず，また目的関数も複数とることができないため，工夫が必要となります．

PySIMPLE では目的関数に添字が残っていたり，複数の目的関数を設定するとエラーとなります．：

```
>>> p = Problem()
>>> i = Element(value=[1,2], name='i')
>>> x = Variable(index=i, name='x')
>>> p += x[i]  # 添字が残っている
pysimple.error.SimpleError: objective cannot be defined with index
>>> p += x[1]  # 目的関数 (1 つ目)
>>> p += x[2]  # 目的関数 (2 つ目)
pysimple.error.SimpleError: objective can only be assigned once
```

多目的最適化の例として，制約 x ≦ 2，y ≦ 2，z ≦ 2，x+y+z ≧ 4 の元で，x + y，y + z，z + x のそれぞれを最小化したいとします．このとき目的関数をどのように設定したらよいでしょうか．：

```
x = Variable(ub=2)
y = Variable(ub=2)
z = Variable(ub=2)

p = Problem()
p += x + y + z >= 4
p += ?, ' 目的関数'
```

多目的最適化を実現する方法の 1 つは，多段階で最適化を行うというものです．これを汎用的に実装したものが以下となります．：

```
objs = x + y, y + z, z + x
for obj in objs:
    p += obj
    p.solve()
    Printf('x: {:.1f} y: {:.1f} z: {:.1f}', x.val, y.val, z.val)
    p += obj <= p.objective.val + 1e-5
```

上記のモデルでは次のようなステップで求解を行っております.

1. 目的関数に x + y を設定して求解 (x=1, y=1, z=2)
2. 問題に制約「x + y ≦ 2(=目的関数値)」を追加
3. 目的関数に y + z を設定して求解 (x=2, y=0, z=2)
4. 問題に制約「y + z ≦ 2(=目的関数値)」を追加
5. 目的関数に z + x を設定して求解 (x=2, y=0, z=2)

今回のモデルでは，ステップ 1 でいわゆる「パレート最適」となっており，最適解が複数存在しています．また，この方法の場合，目的関数の設定順序によって最終的な解が変わる可能性がある点に注意しましょう．

多目的最適化を実現する別の方法は，目的関数ごとに重みをつけて同時に扱う方法です．これを実装したものが以下となります．:

```
for w1, w2 in (1, 1), (1, 2), (2, 1):
    p += (x + y) + w1*(y + z) + w2*(z + x)
    p.solve()
    Printf('x: {:.1f} y: {:.1f} z: {:.1f}', x.val, y.val, z.val)
```

重み w1, w2 の値によって変数の値は次のようになりました.

- w1=1, w2=1 のとき x=1.3, y=1.3, z=1.3
- w1=1, w2=2 のとき x=1, y=2, z=1
- w1=2, w2=1 のとき x=2, y=1, z=1

今回は重みによって変数の値が変わる様子を確かめるために複数求解を行いましたが，重みが決まっている場合は必要ありません．一方で，この重みの決め方が難しいのですが，重みは目的関数ごとのトレードオフ割合と考えることができます.

例えば，目的関数が「コスト (万円) + w*移動距離 (km)」だった場合，w=1 の場合は「コスト 1 万円の削減」と「移動距離 1km の短縮」を同価値に，w=10 の場合は「コスト 10 万円の削減」と「移動距離 1km の短縮」を同価値に考えていることになります．単位が異なる目的関数を同時に考える場合は取りうる範囲に十分注意する必要があり，必要に応じて正規化を施します.

3.8.18 コールバック関数

コールバックとは最適化計算が定期的に呼び出すユーザ関数のことです．ユーザが最適化の状態を問い合わせ，最適化計算の挙動を制御することができます．

分枝限定法のユーザ停止

PySIMPLE では分枝限定法の停止を制御するコールバック関数を設定することができます．ユーザは Problem インスタンスのメソッド *setCallback* の引数 `mip_terminate` にユーザの定義した関数（コールバック関数）を渡すと，その関数が分枝限定法の計算内で呼び出されます．

コールバック関数は一つの引数をとり，bool 型を返す関数を想定します．以下はユーザが定義するコールバック関数の関数定義です．:

```
def mip_terminate(solver: dict[str, int | float]) -> bool:
```

True を返すと分枝限定法を停止し，False を返すと継続して実行されます．コールバック関数の引数には，本関数が呼び出された時点の最適化計算の情報が辞書型に格納されています．以下は，引数に渡ってくる辞書型に格納されている最適化計算の情報一覧です．

キー	内容
ElapsedTime	経過時間
RelativeGap	相対ギャップ
AbsoluteGap	絶対ギャップ
Objective	目的関数値
SolutionCount	実行可能解の数
TotalMemory	メモリ消費量 (MiB)
MemoryAvailable	利用可能メモリ量 (MiB)
PartialProblemCount	部分問題の数 (ノード数)

以下はコールバック関数の記述例です．:

```
def func(solver: dict[str, int | float]) -> bool:
    # 実行可能解の個数が 1 以上になったら分枝限定法を停止する
    if solver['SolutionCount'] >= 1:
        return True
    return False
```

上記で定義した関数 func を Problem インスタンス problem に設定します．:

```
problem.setCallback(mip_terminate=func)
```

以下では停止条件を設定せず，最適化計算の経過情報をカスタマイズして出力しています．:

```
def func(solver: dict[str, int | float]) -> bool:
    print(solver['Objective'], solver['RelativeGap'], solver['AbsoluteGap'])
    return False
```

コールバック関数で停止した場合，ステータス番号 NUOPT 31 あるいは NUOPT 32 で最適化計算が終了します．例えば，ステータス番号が NUOPT 31 の場合は以下のように出力されます．:

```
[Result]
STATUS                                              NON_OPTIMAL
ERROR_TYPE  (NUOPT 31) B&B terminated by user (with feas.sol.).
```

以下は注意事項です．

- 実行可能解が得られていない場合，AbsoluteGap / RelativeGap は +inf の値になっています．
- 実行可能解が得られていない場合，Objective は最大化問題の場合 -inf，最小化問題の場合 +inf の値になっています．
- 最適化計算情報の内，MemoryAvailable は Windows 版のみ有効です．

3.8.19 型ヒントの使い方

バージョン 1.6.0 から型ヒントに対応しました．pysimple.typing モジュールで import 可能なオブジェクトは 型ヒント をご覧ください．

基本的な使い方

pysimple モジュールから直接 import できるクラスは，そのまま型ヒントに用いることができます．:

```
from pysimple import Element

def element_name(i: Element) -> str:
    return i.name

i = Element(value=[1, 2])
print(element_name(i))  # i
```

pysimple モジュールから直接 import できない一部のクラスは pysimple.typing から import できます．:

```
from pysimple import Condition
from pysimple.typing import Cond

def cond_name(c: Cond) -> str:
    return c.name
```

(次のページに続く)

(前のページからの続き)

```
i = Element(value=[1, 2])
c = Condition(i, i>1)  # c: Cond
print(cond_name(c))    # (i, (i>1))
```

pysimple.typing には基底クラスやエイリアスも含まれています. :

```
from pysimple import Element, Condition
from pysimple.typing import ELEMENT

def ELEMENT_name(e: ELEMENT) -> str:
    return e.name

i = Element(value=[1, 2])
c = Condition(i, i>1)   # c: Cond
print(ELEMENT_name(i))  # i
print(ELEMENT_name(c))  # (i, (i>1))
```

IntegerVariable や *BinaryVariable* は *Variable* 型になることにご注意ください. :

```
from pysimple import Variable, IntegerVariable

def variable_name(v: Variable) -> str:
    return v.name

def ivariable_name(v: IntegerVariable) -> str:
    return v.name

x = IntegerVariable()  # x: Variable
variable_name(x)   # OK
#ivariable_name(x)  # NG
```

高度な使い方

次の例では *Set.__getitem__* の戻り値の型を考えています. :

```
from typing import reveal_type
from pysimple import Set

# Set.__getitem__(self, key: int) -> DType | Key
I = Set(value=[1, 2])
i0 = I[0]  # int

IJ = Set(value=[(1, 3), (1, 4), (2, 3)])
ij0 = IJ[0]  # tuple[int, int]
```

`Set.__getitem__(int)` の戻り値は *DType* | *Key* ですが，実際には Set の次元が 1 のときは *DType*，2 以上のときは *Key* となります．これは型チェックでは判定できないため，assert などによって type narrowing を行うことで，手動で型を絞り込むことができます．:

```
reveal_type(i0)
reveal_type(ij0)

# type narrowing
assert isinstance(i0, int)
assert isinstance(ij0, tuple)

reveal_type(i0)
reveal_type(ij0)
```

このような関数，プロパティは他にも *Set.next*, *Set.prev*, *Parameter.__abs__*, *Parameter.__ceil__*, *Parameter.__floor__*, dual 属性，*pysimple.func* などがあります．

次に，チュートリアルの **集合・添字** のようなパターンを見てみましょう．:

```
from pysimple import Problem, Element, Parameter, Variable

i = Element(value=[0, 1])
j = Element(value=['重油', 'ガス'])

costX = Parameter(index=i, value={0: 180, 1: 160})
norma = Parameter(index=j, value={'重油': 12, 'ガス': 24})
x = Variable(lb=0, ub=5, index=i)

problem = Problem()
problem += costX[0]*x[0] + costX[1]*x[1]
problem += 6*x[0] +   x[1] >= norma['重油']
problem += 4*x[0] + 6*x[1] >= norma['ガス']
```

ここでは `costX[0]*x[0]` の部分で型チェッカーがエラーを検出します．これは Parameter は値に数値または文字列をとることができできる一方で，Variable は文字列との積は定義されていないためです．

Parameter の値の型を指定する方法は現在，提供されていないため，例えば `typing.cast` を使うことで型エラーを回避できます．:

```
from typing import cast
:
problem += cast(float, costX[0])*x[0] + cast(float, costX[1])*x[1]
```

第4章　サンプル

PySIMPLE は様々な定式化の例を同梱しています.

4.1 チュートリアル

チュートリアルで扱ったモデル一覧です．具体的な問題は **チュートリアル** をご確認ください.

`sample.tutorial.oil1`(**kwds: *Any*) → None

油田問題 (目的関数・変数・制約)

`sample.tutorial.oil2`(**kwds: *Any*) → None

油田問題 (定数)

`sample.tutorial.oil3`(**kwds: *Any*) → None

油田問題 (集合・添字)

`sample.tutorial.oil4`(**kwds: *Any*) → None

油田問題 (集約・複数の添字)

`sample.tutorial.oil5`(**kwds: *Any*) → None

油田問題 (式)

`sample.tutorial.oil6`(**kwds: *Any*) → None

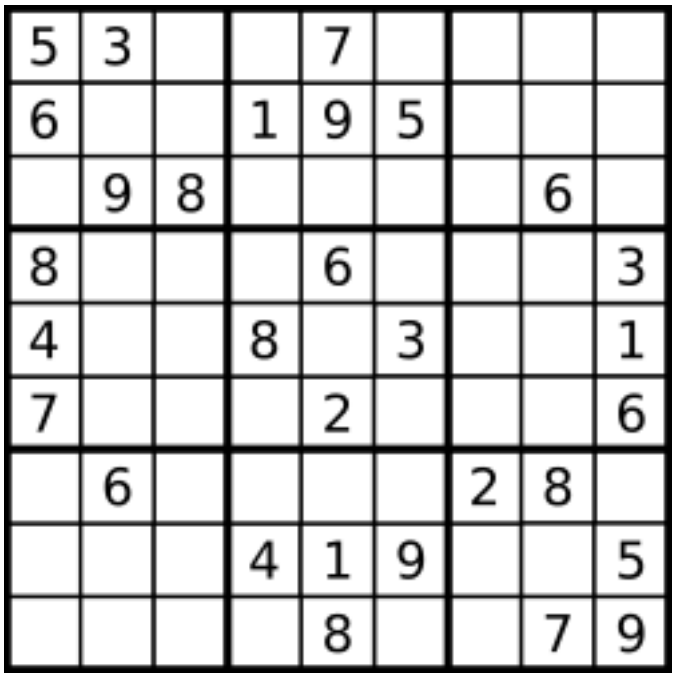
油田問題 (整数変数)

`sample.tutorial.oil7`(**kwds: *Any*) → None

油田問題 (結果出力関数)

4.2 数独

次の数独の問題を PySIMPLE で記述してみましょう.



数独とは 3 × 3 のブロックに区切られた 9 × 9 の正方形の枠内に次を満たすように 1〜9 までの数字を入れるパズルです.

- 空いているマスに 1〜9 のいずれかの数字を入れる
- 各値は縦の各列に 1 つずつ入る
- 各値は横の各行に 1 つずつ入る
- 各値は太線で囲まれた 3 × 3 のブロック内に 1 つずつ入る
- 初期条件を満たす

数独の変数は，各マスについて，各値が「入る」もしくは「入らない」という状態を表現できるものとなります．ここでは，入る場合は 1，入らない場合は 0 を取るような変数 $x_{値,\ マス}$ を導入します．値は 1 から 9 を，マスは左上の (1, 1) から右下の (9,9) をとります.

このとき上記の制約はどのように表すことができるでしょうか．各制約について見ていきましょう.

空いているマスに 1〜9 のいずれかの数字を入れる

マス (1,1) を例に考えると，このマスには 1 から 9 のいずれかの値が入ります．言い換えると，$x_{1,(1,1)}, \cdots, x_{9,(1,1)}$ のうちちょうど 1 つが 1 となっている，すなわち，これらを足すと必ず 1 になると考えることができます.

$$x_{1,(1,1)} + x_{2,(1,1)} + \cdots + x_{9,(1,1)} = 1$$

これを値 $v \in Val = \{1, .., 9\}$ と $\sum$ を用いて表現すると次のようになります.

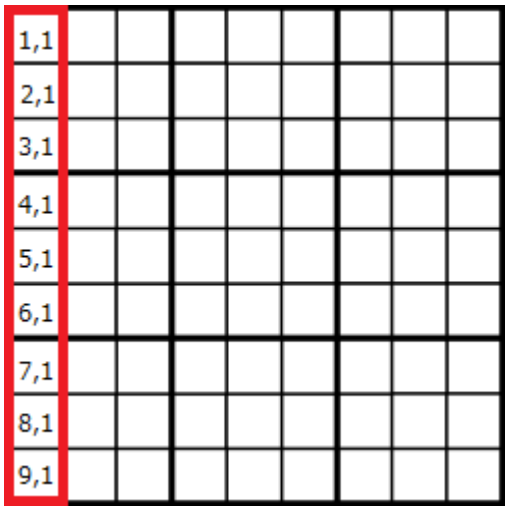
$$\sum_{v \in Val} x_{v,(1,1)} = 1$$

この制約が各マス (r,c) で成立しているので，1 つ目の制約は $r \in Row = \{1, .., 9\}, c \in Col = \{1, .., 9\}$ を用いて次のように記述できます.

$$\sum_{v \in Val} x_{v,(r,c)} = 1, \quad \forall r \in Row, c \in Col$$

各値は縦の各列に 1 つずつ入る

値 1 と 1 列目を例に考えてみましょう．値 1 は 1 列目のいずれかに入ります．1 列目は (1,1), .., (9,1) のマスから成っているので，このうちちょうど 1 つが 1 となります．



これは次のように表現することができます．

$$x_{1,(1,1)} + x_{1,(2,1)} + \cdots + x_{1,(9,1)} = 1$$

すなわち，

$$\sum_{r \in Row} x_{1,(r,1)} = 1$$

となります．これを任意の値 v と任意の列 c について考えると，次のように記述できます．

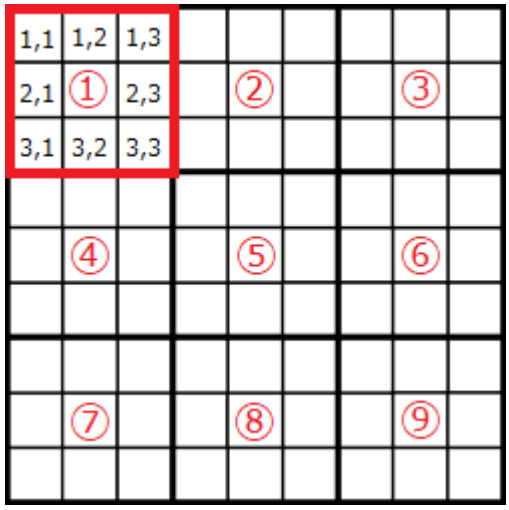
$$\sum_{r \in Row} x_{v,(r,c)} = 1, \quad \forall v \in Val, c \in Col$$

各値は横の各行に 1 つずつ入る

列のときと同様に次のように表現することができます．

$$\sum_{c \in Col} x_{v,(r,c)} = 1, \quad \forall v \in Val, r \in Row$$

各値は太線で囲まれた 3 × 3 のブロック内に 1 つずつ入る



例えば，値 1 が左上のブロック ① に入るという制約は次のように表現することができます.

$$x_{1,(1,1)} + x_{1,(1,2)} + x_{1,(1,3)} + x_{1,(2,1)} + \cdots + x_{1,(3,3)} = 1$$

これは，Block1={(1,1), .., (3,3)} という集合を用意すると，

$$\sum_{(r,c) \in Block1} x_{1,(r,c)} = 1$$

と表現することができます.

他のブロックについても汎用的に記述するにはどうすればよいでしょうか．準備として，Blocks={(①, (1,1)), .., (①, (3,3)), (②, (1,4)), .., (⑨, (9,9))} という集合を用意すると，先ほどの制約は,

$$\sum_{\substack{(r,c) \\ (1,r,c) \in Blocks}} x_{1,(r,c)} = 1$$

と表現できます．任意の値 v と任意のブロック $n \in$ Blocks(0)={①, .., ⑨} について考えると，

$$\sum_{\substack{(r,c) \\ (n,r,c) \in Blocks}} x_{v,(r,c)} = 1, \quad \forall v \in Val, n \in Blocks(0)$$

と記述できます．ここで Blocks(0) は三次元集合 Blocks の一次元目を表します.

初期条件を満たす

初期値集合 Init = {(5,1,1), (6,2,1), .., (9,9,9)} を用意すると，次のように表現することができます.

$$x_{v,(r,c)} = 1, \quad (v,r,c) \in Init$$

以上のことを反映し，汎用化させた結果は以下のようになります.

集合	
$Val = \{1, .., 9\}$	値集合
$Row = \{1, .., 9\}$	行集合
$Col = \{1, .., 9\}$	列集合
$Blocks = \{(1,1,1), .., (1,3,3), (2,1,4), .., (9,9,9)\}$	ブロック集合
$Init = \{(5,1,1), (6,2,1), .., (9,9,9)\}$	初期値集合

$\mathbf{0-1}$ 整数変数	
$x_{v,r,c}$	値 v がマス (r,c) に入るとき 1，入らないとき 0 をとる変数

制約	
$\sum_{v \in Val} x_{v,r,c} = 1, \quad \forall r \in Row, c \in Col$	各マスにはいずれかの値を入れる
$\sum_{r \in Row} x_{v,r,c} = 1, \quad \forall v \in Val, c \in Col$	各値は縦の各列に 1 つずつ入る
$\sum_{c \in Col} x_{v,r,c} = 1, \quad \forall v \in Val, r \in Row$	各値は横の各行に 1 つずつ入る
$\sum_{\substack{(r,c) \\ (n,r,c) \in Blocks}} x_{v,r,c} = 1, \quad \forall v \in Val, n \in Blocks(0)$	各値は 3×3 のブロック内に 1 つずつ入る
$x_{v,r,c} = 1, \quad (v,r,c) \in Init$	初期条件を満たす

これを PySIMPLE で記述すると，次のようになります．:

```
from pysimple import Problem, Set, Element, BinaryVariable, Sum, Parameter

Vals = Rows = Cols = Set(value=range(1, 10))
v = Element(set=Vals)
r = Element(set=Rows)
c = Element(set=Cols)
Blocks = Set(value=[(s+1, s//3*3+t//3+1, s%3*3+t%3+1) for s in range(9) for t in␣
↪range(9)])
b = Element(set=Blocks)  # b(0) は Block 番号
print(Blocks)

Init = Set(value=[(5, 1, 1), (6, 2, 1), (8, 4, 1), (4, 5, 1), (7, 6, 1), (3, 1, 2),␣
↪(9, 3, 2), (6, 7, 2), (8, 3, 3), (1, 2, 4), (8, 5, 4), (4, 8, 4), (7, 1, 5), (9, 2,
 5), (6, 4, 5), (2, 6, 5), (1, 8, 5), (8, 9, 5), (5, 2, 6), (3, 5, 6), (9, 8, 6), (2,␣
↪7, 7), (6, 3, 8), (8, 7, 8), (7, 9, 8), (3, 4, 9), (1, 5, 9), (6, 6, 9), (5, 8, 9),␣
↪(9, 9, 9)])
init = Element(set=Init)

prob = Problem(name='数独')
x = BinaryVariable(index=(v,r,c))  # 値 v が (r,c) に入るか

prob += Sum(x[v,r,c], v) == 1          # 各マスにはいずれかの値
prob += Sum(x[v,r,c], r) == 1          # 各値は各列のいずれか
prob += Sum(x[v,r,c], c) == 1          # 各値は各行のいずれか
prob += Sum(x[v,b(1,2)], b(1,2)) == 1  # 各値は各ブロックのいずれか
prob += x[init] == 1                   # 初期条件

print(prob)
prob.solve()

print(x[x[v,r,c].val==1])

inittable = Parameter(index=(v,r,c), value=dict.fromkeys(Init, 1))
resulttable = dict(x.val.items())  # dump as dict
showsudoku(Rows, Cols, Vals, inittable)
showsudoku(Rows, Cols, Vals, resulttable)
```

ここで showsudoku は数独のテーブルを整形して出力するための次のような関数です．：

```
def showsudoku(Rows: Set, Cols: Set, Vals: Set, table: dict) -> None:
    from sys import stdout

    for r, in Rows:
        if r in (1, 4, 7):
            stdout.write("+-------+-------+-------+\n")
        for c, in Cols:
```

(次のページに続く)

(前のページからの続き)

```
            for v, in Vals:
                if table[v,r,c] == 1:
                    break
            else:
                v = ' '
            if c in (1, 4, 7):
                stdout.write("| ")
            stdout.write(str(v) + " ")
        stdout.write("|\n")
    stdout.write("+-------+-------+-------+\n")
```

モデルを実行させると，最後に次のような出力が得られ，問題が解けていることが分かります．:

```
+-------+-------+-------+
| 5 3   |   7   |       |
| 6     | 1 9 5 |       |
|   9 8 |       |   6   |
+-------+-------+-------+
| 8     |   6   |     3 |
| 4     | 8   3 |     1 |
| 7     |   2   |     6 |
+-------+-------+-------+
|   6   |       | 2 8   |
|       | 4 1 9 |     5 |
|       |   8   |   7 9 |
+-------+-------+-------+
+-------+-------+-------+
| 5 3 4 | 6 7 8 | 9 1 2 |
| 6 7 2 | 1 9 5 | 3 4 8 |
| 1 9 8 | 3 4 2 | 5 6 7 |
+-------+-------+-------+
| 8 5 9 | 7 6 1 | 4 2 3 |
| 4 2 6 | 8 5 3 | 7 9 1 |
| 7 1 3 | 9 2 4 | 8 5 6 |
+-------+-------+-------+
| 9 6 1 | 5 3 7 | 2 8 4 |
| 2 8 7 | 4 1 9 | 6 3 5 |
| 3 4 5 | 2 8 6 | 1 7 9 |
+-------+-------+-------+
```

このモデルを PySIMPLE で記述するためのポイントをいくつか示します．:

```
Blocks = Set(value=[(s+1, s//3*3+t//3+1, s%3*3+t%3+1) for s in range(9) for t in␣
↪range(9)])
b = Element(set=Blocks)  # b(0) は Block 番号
```

(次のページに続く)

(前のページからの続き)

```
prob += Sum(x[v,b(1,2)], b(1,2)) == 1  # 各値は各ブロックのいずれか
```

ここではまず，三組み Blocks を作成しています．print(Blocks) をすると，上記の要素から成ることが確認できます．b は三組みの任意の 1 つを表す添字です．ブロックの制約 $\sum_{\substack{(r,c)\\(n,r,c)\in Blocks}} x_{v,r,c} = 1$ では三組みの後ろ 2 つについての和をとる必要があります．これは次のように表現することができます．

$$\sum_{\substack{b(1,2)\\b\in Blocks}} x_{v,b(1,2)} = 1, \quad \forall v \in Val, b(0) \in Blocks(0)$$

ここで b(1,2) は添字 b の 1 番目と 2 番目 (0 始まり) を表します．

同様に初期条件も三組を考えて次のように表現します．:

```
Init = Set(value=[(5, 1, 1), (6, 2, 1), .., (9, 9, 9)])
init = Element(set=Init)
prob += x[init] == 1                    # 初期条件
```

```
print(x[x[v,r,c].val==1])
```

この部分では求解後に値が入った組合せを確認しています．三組だけを取り出したい場合は (x[v,r,c].val==1).set で OK です．

```
inittable = Parameter(index=(v,r,c), value=dict.fromkeys(Init, 1))
resulttable = dict(x.val.items())  # dump as dict
showsudoku(Rows, Cols, Vals, inittable)
showsudoku(Rows, Cols, Vals, resulttable)
```

この部分では数独のテーブルを整形して出力しています．showsudoku はキーが (1,1,1), .., (9,9,9)，値が 0 か 1 である辞書を受け取る関数です．

inittable ではまず，dict.fromkeys(Init, 1) で集合から辞書に変換しています．このままでは定義域が不足しているので，定義域が (v,r,c) の Parameter に value として渡すことで目的の辞書を作成しています．Parameter は value で渡された部分はその値を，それ以外は値を 0 とする辞書を作成するためです．また，resulttable では求解後の変数の値を辞書に dump しています．

sample.sudoku.**showsudoku**(*Rows:* pysimple.element.Set, *Cols:* pysimple.element.Set, *Vals:* pysimple.element.Set, *table:* *dict*) → None

sample.sudoku.**sudoku**(***kwds:* *Any*) → None

数独

4.3 例題集

C++SIMPLE 例題集 に対応したモデルの PySIMPLE による実装を同梱しています．具体的な問題は C++SIMPLE 例題集を確認ください．

sample.reidaishu.**p2010_mixture**(**kwds: *Any*) → None

配合問題

sample.reidaishu.**p2020_transport2**(**kwds: *Any*) → None

輸送問題

sample.reidaishu.**p2030_multiplan2**(**kwds: *Any*) → None

多期間計画問題

sample.reidaishu.**p2040_DEA**(**kwds: *Any*) → None

包絡分析法（DEA）モデル

sample.reidaishu.**p2050_knapsack2**(**kwds: *Any*) → None

ナップサック問題

sample.reidaishu.**p2060_cover2**(**kwds: *Any*) → None

集合被覆問題

sample.reidaishu.**p2070_maxflow2**(**kwds: *Any*) → None

最大流問題

sample.reidaishu.**p2071_maxflow3**(**kwds: *Any*) → None

最大流問題 (Matrix)

sample.reidaishu.**p2072_mincut**(**kwds: *Any*) → None

最小カット問題

sample.reidaishu.**p2080_mincost2**(**kwds: *Any*) → None

最小費用流問題

sample.reidaishu.**p2090_multiflow2**(**kwds: *Any*) → None

多品種流問題

sample.reidaishu.**p2100_median2**(**kwds: *Any*) → None

p メディアン問題

sample.reidaishu.**p2110_center2**(**kwds: *Any*) → None

p センター問題

sample.reidaishu.**p2120_TSP3**(**kwds: *Any*) → None

巡回セールスマン問題

sample.reidaishu.**p2132_fieldassign2**(*method*: pysimple.options.Options.Method = *Method.AUTO*, **kwds: *Any*) → None

基礎的なマス埋め割当問題

sample.reidaishu.**p2132_fieldassign3**(*method:* pysimple.options.Options.Method = *Method.WCSP*, ***kwds: Any*) → None

基礎的なマス埋め割当問題 (wcsp)

sample.reidaishu.**p2133_jobassign3**(*method:* pysimple.options.Options.Method = *Method.AUTO*, ***kwds: Any*) → None

仕事割当問題

sample.reidaishu.**p2133_jobassign4**(*method:* pysimple.options.Options.Method = *Method.WCSP*, ***kwds: Any*) → None

仕事割当問題 (wcsp)

sample.reidaishu.**p2140_QAP**(***kwds: Any*) → None

二次割当問題

sample.reidaishu.**p2150_FPP**(***kwds: Any*) → None

設備計画問題

sample.reidaishu.**p2160_leastsquare2**(***kwds: Any*) → None

最小二乗問題

sample.reidaishu.**p2170_portfolio1**(***kwds: Any*) → None

ポートフォリオ最適化問題

sample.reidaishu.**p2230_maxcut2**(***kwds: Any*) → None

隣接行列 (最大カット問題)

sample.reidaishu.**p2260_pseudoinverse**(***kwds: Any*) → None

ムーア・ペンローズ一般逆行列

4.4 列生成法

列生成法の基底クラスです．理論については以下もご確認ください．

- 列生成法という技法 - 定式化技法集
- ナップサック問題と列生成法 - 定式化設計集

4.4.1 ビンパッキング問題

ビンパッキング問題を列生成法で解くサンプルです．

class sample.bin_packing.**BinPacking**(*binsize: int*, *sizevalue: dict[str, int]*)

ベースクラス: *sample.column_generator.ColumnGenerator*

ビンパッキング問題

create_init_pattern() → None

set self.pattern, self.b, self.c

create_new_pattern(*lmbval:* pysimple.table.Table, *, *silent: bool = True*) → tuple[*pysimple.table.Table*, *pysimple.table.Table*]

ビンに収まるアイテムの組合せを 1 つ生成する

visualize(*zval:* pysimple.table.Table) → None

結果表示

sample.bin_packing.**at_once**(*binsize: int*, *sizevalue: dict[str, int]*, *, *silent: bool = True*) → None

一度に全部解く

sample.bin_packing.**column_generation**(*binsize: int*, *sizevalue: dict[str, int]*) → None

列生成で解く

sample.bin_packing.**create_init_data**(*, *binsize: int*, *N: int*) → tuple[int, dict[str, int]]

binsize: ビンサイズ, N: アイテム種類数

sample.bin_packing.**enumerate_all_patterns**(*binsize: int*, *sizevalue: dict[str, int]*) → None

全列挙で解く

4.4.2 カッティングストック問題

カッティングストック問題を列生成法で解くサンプルです.

class sample.cutting_stock.**CuttingStock**(*L: int*, *lenvalue: dict[str, int]*, *reqvalue: dict[str, int]*)

ベースクラス: *sample.column_generator.ColumnGenerator*

カッティングストック問題

create_init_pattern() → None

set self.pattern, self.b, self.c

create_new_pattern(*lmbval:* pysimple.table.Table, *, *silent: bool = True*) → tuple[*pysimple.table.Table*, *pysimple.table.Table*]

母材に収まる木材の組合せを 1 つ生成する

select_pattern(*, *vtype=<class 'int'>*, *silent: bool = True*) → tuple[*pysimple.table.Table*, *pysimple.table.Table*]

生成されたパターンから需要を満たす組合せを選択する

visualize(*zval:* pysimple.table.Table) → None

結果表示

sample.cutting_stock.**create_init_data**(*, *L: int*, *N: int*) → tuple[int, dict[str, int], dict[str, int]]

L: 母材の長さ, N: 切り出す木材の種類の数

4.4.3 Vehicle Routing Problem

Vehicle Routing Problem を列生成法で解くサンプルです.

class sample.vehicle_routing.**VehicleRouting**(*capacity: int*, *volume: dict[str, int]*, *dis: dict[tuple[str, str], float]*)

ベースクラス: *sample.column_generator.ColumnGenerator*

Vehicle Routing Problem

create_init_pattern() → None

倉庫 X と店舗 i を往復するルート

create_new_pattern(*lmbval:* pysimple.table.Table, *, *silent: bool = True*) → tuple[*pysimple.table.Table*, *pysimple.table.Table*]

Sum(pattern[p,i]*lmb[i], i) < rd[p] となる p を見つける

update_pattern(*zval:* pysimple.table.Table) → None

update self.pattern, self.c

visualize(*zval:* pysimple.table.Table) → None

結果表示

sample.vehicle_routing.**create_init_data**(*, *N: int*) → tuple[int, dict[str, int], dict[tuple[str, str], float]]

N: 店舗数

class sample.column_generator.**ColumnGenerator**

列生成法のための抽象基底クラス

abstract create_init_pattern(*, *pattern: dict*, *b: float | dict = 1*, *c: float | dict = 1*) → None

set self.pattern, self.b, self.c

create_lambda(*, *silent: bool = True*) → *pysimple.table.Table*

select_pattern の双対問題

abstract create_new_pattern(*lmbval:* pysimple.table.Table, *, *silent: bool = True*) → tuple[*pysimple.table.Table*, *pysimple.table.Table*]

組合せを 1 つ生成する

property iternum

property patternnum

select_pattern(*, *vtype=<built-in function bin>*, *silent: bool = True*) → tuple[*pysimple.table.Table*, *pysimple.table.Table*]

生成されたパターンから b を満たす組合せを選択する

solve(*, *eps: float = 0.0*, *maxiter: int = 100*) → *pysimple.table.Table*

列生成法で問題を解く

update_pattern(*zval:* pysimple.table.Table) → None

update self.pattern, self.c

以下のコマンドを実行すると上記のすべての問題がサイレントモードで実行されます．:

```
$ python -m pysimple.sample
```

以下のコマンドを実行するとチュートリアルのすべての問題が実行されます．:

```
$ python -m pysimple.sample.tutorial
```

問題ごとに実行することもできます．:

```
$ python -m pysimple.sample.tutorial oil7
```

以下のコマンドを実行すると数独の問題が実行されます．:

```
$ python -m pysimple.sample.sudoku
```

問題ごとに実行することもできます．:

```
$ python -m pysimple.sample.sudoku sudoku
```

以下のコマンドを実行すると例題集のすべての問題が実行されます．:

```
$ python -m pysimple.sample.reidaishu
```

問題ごとに実行することもできます．:

```
$ python -m pysimple.sample.reidaishu p2010_mixture
```

以下のコマンドを実行するとビンパッキング問題を列生成法で実行します．:

```
$ python -m pysimple.sample.bin_packing
```

以下のコマンドを実行するとカッティングストック問題を列生成法で実行します．:

```
$ python -m pysimple.sample.cutting_stock
```

以下のコマンドを実行すると Vehicle Routing Problem を列生成法で実行します．:

```
$ python -m pysimple.sample.vehicle_routing
```

第5章　API ドキュメント

PySIMPLE の API ドキュメントです.

5.1 クラス

コンストラクタが公開されているクラスは，すべての引数をキーワード引数として呼び出さなければなりません．複数の引数を持たせる場合，キーワード引数の順序は任意です.

インスタンス間に定義されている演算については 演算 を参照してください.

5.1.1 集合

class pysimple.**Set**(*name: str | None = None*, *value: Iterable | None = None*, *dim: int | None = None*)

ベースクラス: collections.abc.Set[tuple[int | float | str, ...]], pysimple.util.Named

集合を表すクラスです．Set はすべて順序を持っています．オブジェクトの生成後は要素を変更できません.

パラメータ

- **name** (*str*) -- 省略した場合は自動で与えられます．オブジェクトの生成後に変更できます.
- **value** (Iterable[*DType* | *Key*]) -- イテラブルなオブジェクトを指定します．多次元の場合は tuple の列を与えます．省略した場合は空集合が作成されます.
- **dim** (*positive integer*) -- 省略できます．多次元の空集合を作成する場合は明示的に指定する必要があります.

例外

- **TypeError** -- {} keyword argument 'dim' must be positive integer ({} given)
- **TypeError** -- {} object is not iterable
- **ValueError** -- dimension mismatch
- **ValueError** -- empty data: {}
- **ValueError** -- illegal type data: {}

サンプル

```
>>> Set(value=[1,2], name='I')
Set(name='I', value=[1, 2])
>>> Set(value=range(3))
Set(value=[0, 1, 2])
>>> Set(value='XYZ')
Set(value=['X', 'Y', 'Z'])
>>> Set(value=[(1,3), (1,4), (2,3)])
Set(dim=2, value=[(1, 3), (1, 4), (2, 3)])
>>> Set()
Set()
>>> Set(dim=2)
Set(dim=2)
```

__call__(**slc: int*) → *Set*

各次元を射影した集合を返します.

パラメータ ***slc** (*non-negative integers*) -- 射影する次元の列です (0 始まり). 1 つ以上必要です.

戻り値の型 *Set*

例外

- **TypeError** -- slice of {} takes at least 1 argument (0 given)
- **TypeError** -- slice of {} indices must be non-negative integers
- **IndexError** -- slice index out of range

サンプル

```
>>> I = Set(value=[(1,3), (1,4), (2,3)], name='I')
>>> I(0)
Set(name='I(0)', value=[1, 2])
>>> I(1)
Set(name='I(1)', value=[3, 4])
>>> I(2)
IndexError: slice index out of range
>>> I(1, 0, 1)
Set(name='I(1,0,1)', dim=3, value=[(3, 1, 3), (4, 1, 4), (3, 2, 3)])
>>> I()
TypeError: slice of 'I' takes at least 1 argument (0 given)
```

__contains__(*key: DType | Key*) → bool

key が要素に含まれているかを返します.

パラメータ **key** (*DType* | *Key*) --

戻り値の型 bool

注釈: Element のうち，集合に含まれる部分を表現したい場合は比較演算子を用いてください.

参考:

pysimple.__lt__, pysimple.__gt__

サンプル

```
>>> I = Set(value=[1,2], name='I')
>>> 0 in I
False
>>> 1 in I
True
>>> i = Element(value=[1,2], name='i')
>>> i in I
TypeError: unsupported operand type(s) for 'Element' in 'Set', use 'i < I'␣
↪instead of 'i in I'
>>> J = Set(value=[(1,3), (1,4), (2,3)], name='J')
>>> (2,3) in J
True
>>> (2,4) in J
False
```

__getitem__(*key: int* | *ELEMENT* | *slice*) → DType | Key | *Set* | *Parameter*

Set の key 番目の要素を返します．key には負数やスライスも指定できます.

パラメータ **key** (int or *ELEMENT* or slice) --

戻り値 key が int の場合は Set の要素を，*ELEMENT* の場合は Parameter を，slice の場合は Set を返します.

戻り値の型 *DType* or *Key* or Parameter or Set

例外

- **IndexError** -- Set index out of range
- **TypeError** -- Set indices must be integers, not {}
- **TypeError** -- cannot apply {} to multidimensional Set

参考:

pysimple.Set.next, *pysimple.Set.prev*, *pysimple.Set.index*

サンプル

```
>>> S = Set(value='XY', name='S')
>>> S[-3]
IndexError: Set index out of range
>>> S[-2]
'X'
>>> S[-1]
'Y'
>>> S[0]
'X'
>>> S[1]
'Y'
>>> S[2]
IndexError: Set index out of range
>>> S['foo']
TypeError: Set indices must be integers, not str
>>> i = Element(value=[-2, 1], name='i')
>>> S[i]
S[i][-2]='X'
S[i][1]='Y'
>>> I = Set(value=range(5), name='I')
>>> I
Set(name='I', value=[0, 1, 2, 3, 4])
>>> I[2:]
Set(name='I[2:]', value=[2, 3, 4])
```

__iter__() → collections.abc.Iterator[Key]

iter(self) を返します.

戻り値の型 Iterator[*Key*]

サンプル

```
>>> I = Set(value=[1,2])
>>> for i in I:
...     print(i)
...
(1,)
(2,)
>>> IJ = Set(value=[(1,3), (1,4), (2,3)])
>>> for ij in IJ:
...     print(ij)
...
(1, 3)
```

(次のページに続く)

(前のページからの続き)

```
(1, 4)
(2, 3)
```

注釈: 次元に関わらず，値は常にタプルとして返されることに注意してください.

__len__() → int

要素数を返します.

戻り値の型 int

property dim

集合の次元です.

index(*key: DType | Key | ELEMENT*) → int | *Parameter*

key が Set の何番目の要素であるかを返します.

パラメータ **key** (*DType* or *Key* or *ELEMENT*) --

戻り値 key が *DType* または *Key* の場合は int を，*ELEMENT* の場合は Parameter を返します.

戻り値の型 int or *Parameter*

例外 **ValueError** -- {} is not in {}

参考:

pysimple.Set.__getitem__, *pysimple.Set.next*, *pysimple.Set.prev*

サンプル

```
>>> S = Set(value='XY', name='S')
>>> S.index('X')
0
>>> S.index('Y')
1
>>> S.index('Z')
ValueError: Z is not in Set
>>> s = Element(set=S, name='s')
>>> S.index(s)
S.index(s)['X']=0
S.index(s)['Y']=1
>>> t = Element(value='YZ', name='t')
>>> S.index(t)
ValueError: Z is not in Set
```

property name: str

name 属性です．値は文字列の代入により変更できます．

next(*key: DType | Key | ELEMENT*) → DType | Key | *Parameter*

Set における key の次の要素を返します．

パラメータ **key** (*DType* or *Key* or *ELEMENT*) --

戻り値 key が *DType* または *Key* の場合は Set の要素を，*ELEMENT* の場合は Parameter を返します．

戻り値の型 *DType* or *Key* or Parameter

例外

- **KeyError** --
- **IndexError** -- Set index out of range
- **TypeError** -- cannot apply {} to multidimensional Set

参考:

pysimple.Set.__getitem__, *pysimple.Set.prev*, *pysimple.Set.index*

サンプル

```
>>> XYZ = Set(value='XYZ', name='XYZ')
>>> XYZ.next('X')
'Y'
>>> XYZ.next('Y')
'Z'
>>> XYZ.next('Z')
IndexError: Set index out of range
>>> XYZ.next('W')
KeyError: 'W'
>>> xy = Element(value='XY', name='xy')
>>> XYZ.next(xy)
XYZ.next(xy)['X']='Y'
XYZ.next(xy)['Y']='Z'
>>> yz = Element(value='YZ', name='yz')
>>> XYZ.next(yz)
IndexError: Set index out of range
```

prev(*key: DType | Key | ELEMENT*) → DType | Key | *Parameter*

Set における key の前の要素を返します．

パラメータ **key** (*DType* or *Key* or *ELEMENT*) --

戻り値 key が *DType* または *Key* の場合は Set の要素を，*ELEMENT* の場合は Parameter を返します．

戻り値の型 *DType* or *Key* or Parameter

例外

- **KeyError** --
- **IndexError** -- Set index out of range
- **TypeError** -- cannot apply {} to multidimensional Set

参考:

pysimple.Set.__getitem__, *pysimple.Set.next*, *pysimple.Set.index*

サンプル

```
>>> XYZ = Set(value='XYZ', name='XYZ')
>>> XYZ.prev('Z')
'Y'
>>> XYZ.prev('Y')
'X'
>>> XYZ.prev('X')
IndexError: Set index out of range
>>> XYZ.prev('W')
KeyError: 'W'
>>> yz = Element(value='YZ', name='yz')
>>> XYZ.prev(yz)
XYZ.prev(yz)['Y']='X'
XYZ.prev(yz)['Z']='Y'
>>> xy = Element(value='YXY', name='xy')
>>> XYZ.prev(xy)
IndexError: Set index out of range
```

バージョン 1.1.0 で追加: *__getitem__()* に *slice* が使用可能になりました.

5.1.2 添字

class pysimple.**Element**(*name: str | None = None*, *set:* Set | *None = None*, *value: Iterable[DType | Key] | None = None*, *dim: int | None = None*)

添字を表すクラスです.

集合と添字を対応付けます. 集合の要素を直接与えることもできます. set キーワードと value キーワードのいずれか一方を指定する必要があります. 要素を確認する場合は set 属性を参照してください.

パラメータ

- **name** (*str*) -- 省略した場合は自動で与えられます. オブジェクトの生成後に変更できます.

- **set** (Set) -- 対応する集合です．value キーワードと併用できません．
- **value** (*Iterable*) -- Set の value キーワードと同じです．set キーワードと併用できません．
- **dim** (*positive integer*) -- 省略できます．多次元の空の添字を作成する場合は明示的に指定する必要があります．

例外 **TypeError** -- Element can only take either keyword argument 'set' or 'value' (no or both keyword arguments given)

サンプル

```
>>> I = Set(value=[1,2], name='I')
>>> i = Element(set=I, name='i')
>>> i.set
Set(name='I', value=[1, 2])
>>> j = Element(value=[1,2], name='j')
>>> j.set
Set(name='j.set', value=[1, 2])
```

__call__(**slc: int*) → *ElementSlice*

各次元を射影した添字を返します．

パラメータ ***slc** (*non-negative integers*) -- 射影する次元の列です (0 始まり)．1 つ以上必要です．

戻り値の型 *ElementSlice*

例外

- **TypeError** -- slice of {} takes at least 1 argument (0 given)
- **IndexError** -- slice index out of range

サンプル

```
>>> i = Element(value=[(1,3), (1,4), (2,3)], name='i')
>>> i(0).set
Set(name='i.set(0)', value=[1, 2])
>>> i(1).set
Set(name='i.set(1)', value=[3, 4])
>>> i(2).set
IndexError: slice index out of range
>>> i(1,0,1).set
Set(name='i.set(1,0,1)', dim=3, value=[(3, 1, 3), (4, 1, 4), (3, 2, 3)])
>>> i()
TypeError: slice of 'Element' takes at least 1 argument (0 given)
```

property name: *str*

name 属性です．値は文字列の代入により変更できます．

property set: *pysimple.element.Set*

対応する集合を表す属性です．

class pysimple.element.**ElementSlice**

Element のスライスに対して生成されるクラスです．

ElementSlice クラスは *__call__()* がないこと以外は，ほぼ *Element* と同じです．ElementSlice クラスのコンストラクタは公開されません．

class pysimple.condition.**Cond**

Condition 関数などの条件付けによって生成されるクラスです．Cond オブジェクトは添字のように使用することができます．

サンプル

```
>>> ij = Element(value=[(1,3), (1,4), (2,3)], name='ij')
>>> ij(1)!=4
(ij(1)!=4)[ij(1)] in [3]
>>> Condition(ij, ij(1)!=4)
(ij, (ij(1)!=4))[ij] in [(1, 3), (2, 3)]
>>> (ij(0)!=2) & (ij(1)!=4)
((ij(0)!=2)&(ij(1)!=4))[ij(0),ij(1)] in [(1, 3)]
>>> i = Element(value=[1,2,3,4], name='i')
>>> i < [2,4,6,8]
(i<[2, 4, 6, 8])[i] in [2, 4]
```

参考:

pysimple.Condition

Cond クラスのコンストラクタは公開されません．

class pysimple.index.**Index**

添字つきオブジェクトの添字情報を保持するクラスです．

__iter__() → collections.abc.Iterator[pysimple.index.IDXT]

property dim: *int*

添字つきオブジェクトの次元です．

5.1.3 定数

class pysimple.**Parameter**(*name: str = None*, *index: Set | ELEMENT | tuple[Set | ELEMENT, ...] | Index | None = None*, *value: DType | dict = 0*)

定数を表すクラスです.

パラメータ

- **name** (*str*) -- 省略した場合は自動で与えられます．オブジェクトの生成後に変更できます.
- **index** (Set or *ELEMENT* or tuple of them or Index) -- Parameter の添字．省略した場合，添字なしの定数となります.
- **value** (*DType* or dict) -- Parameter の値．*DType* の場合，すべての要素に対して同じ値が適用されます．dict の場合，指定された要素の値が適用されます．オブジェクトの生成後に変更できます.

例外 **KeyError** -- extra key {} in value of Parameter

サンプル

```
>>> i = Element(value=[1,2], name='i')
>>> a = Parameter(index=i, name='a')
>>> a
a[1]=0
a[2]=0
>>> b = Parameter(index=i, value=1, name='b')
>>> b
b[1]=1
b[2]=1
>>> c = Parameter(index=i, value={2: 2}, name='c')
>>> c
c[1]=0
c[2]=2
>>> j = Element(value=[3,4], name='j')
>>> d = Parameter(index=(i,j), name='d')
>>> d
d[1,3]=0
d[1,4]=0
d[2,3]=0
d[2,4]=0
>>> e = Parameter(value=1, name='e')
>>> e
e=1
```

__abs__() → None

Fabs(obj: ObjPrm) -> Num | Parameter

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.fabs (obj) の結果を，それ以外の場合は math.fabs を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

__bool__() → bool

真偽値 bool(self) を返します．index なしの場合，value の真偽値を返します．index ありの場合，TypeError が送出されます.

サンプル

```
>>> a = Parameter(value=3)
>>> bool(a)
True
>>> b = Parameter()
>>> bool(b)
False
>>> i = Element(value=[1,2])
>>> c = Parameter(index=i)
>>> bool(c)
TypeError: unsupported cast type(s) for bool: 'Parameter'
```

__ceil__() → None

Ceil(obj: ObjPrm) -> Num | Parameter

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.ceil (obj) の結果を，それ以外の場合は math.ceil を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

__contains__(*key: DType or Key*) → bool

key がキーに含まれているかを返します.

戻り値の型 bool

__float__() → float

浮動小数への変換 float(self) を返します．index なしの場合，value の浮動小数値を返します．index ありの場合，TypeError が送出されます.

サンプル

```
>>> a = Parameter(value=3)
>>> float(a)
3.0
>>> b = Parameter()
>>> float(b)
0.0
>>> i = Element(value=[1,2])
>>> c = Parameter(index=i)
>>> float(c)
TypeError: unsupported cast type(s) for float: 'Parameter'
```

__floor__() → None

Floor(obj: ObjPrm) -> Num | Parameter

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.floor (obj) の結果を，それ以外の場合は math.floor を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

__format__(*spec: str*) → str

整形した文字列表現 format(self, spec) を返します．index なしの場合，value を spec の書式に整形した文字列を返します．index ありの場合，TypeError が送出されます.

サンプル

```
>>> a = Parameter(value=3.14, name='a')
>>> x = Variable(init=2.72, name='x')
>>> f'{a} {a:.1f} {x.val} {x.val:.1f}'
'a=3.14 a=3.1 x.val=2.72 x.val=2.7'
>>> format(a, '.1f'), format(x.val, '.1f')
('a=3.1', 'x.val=2.7')
>>> i = Element(value=[1, 2], name='i')
>>> b = Parameter(index=i, value=3.14, name='b')
>>> f'{b[i]}'
TypeError: unsupported cast type(s) for format: 'Parameter'
```

__getitem__(*key: ObjPrm | tuple[ObjPrm, ...] |* Index) → DType | *Parameter*

key に対応した添字の Parameter を返します.

パラメータ **key** (*ObjPrm* or tuple of them or Index) --

戻り値 key がすべて *DType* の場合，*DType* を返します．それ以外の場合，Parameter を返します.

戻り値の型 *DType* or *Parameter*

例外

- **KeyError** -- key がキーに存在しない場合
- **KeyError** -- '{}' takes {} dimension index ({} given)

サンプル

```
>>> i = Element(value=[1, 2], name='i')
>>> j = Element(value=[3, 4], name='j')
>>> a = Parameter(index=(i,j), name='a'); a[i,j] = i+j
>>> a[i,j]
a[1,3]=4
a[1,4]=5
a[2,3]=5
a[2,4]=6
>>> a[i,3]
a[1,3]=4
a[2,3]=5
>>> a[2,3]
5
>>> ij = Element(value=[(1,3), (1,4), (2,3)], name='ij')
>>> a[ij]
a[1,3]=4
a[1,4]=5
a[2,3]=5
>>> a[1,5]
KeyError: 'a[1,5]'
>>> b = Parameter(index=j, value={3: 4, 4: 3}, name='b')
>>> a[i, b[j]]
a[i,b[j]][1,3]=5
a[i,b[j]][1,4]=4
a[i,b[j]][2,3]=6
a[i,b[j]][2,4]=5
```

参考:

pysimple.Parameter.get

__int__() → int

整数への変換 int(self) を返します．index なしの場合，value の整数値を返します．index ありの場合，TypeError が送出されます．

サンプル

```
>>> a = Parameter(value=3)
>>> int(a)
3
>>> b = Parameter()
>>> int(b)
0
>>> i = Element(value=[1,2])
>>> c = Parameter(index=i)
>>> int(c)
TypeError: unsupported cast type(s) for int: 'Parameter'
```

__iter__() → collections.abc.Iterator[Key]

iter(self) を返します.

戻り値の型 Iterator[*Key*]

サンプル

```
>>> i = Element(value=[1,2])
>>> a = Parameter(index=i)
>>> for _i in a:
...     print(_i)
...
(1,)
(2,)
>>> ij = Element(value=[(1,3), (1,4), (2,3)])
>>> b = Parameter(index=ij)
>>> for _ij in b:
...     print(_ij)
...
(1, 3)
(1, 4)
(2, 3)
```

注釈: 次元に関わらず，値は常にタプルとして返されることに注意してください.

__len__() → int

キーの要素数を返します.

戻り値の型 int

__round__(*ndigits: int | None = None*) → *Parameter*

値の小数部を ndigits 桁に丸めた値を返します．ndigits が省略された場合，最も近い整数を返します．

パラメータ **ndigits** (*int*) -- 小数部を丸める桁数です．0 や負数の場合，整数部となります．

戻り値の型 *Parameter*

サンプル

```
>>> i = Element(value=[1, 2, 3], name='i')
>>> a = Parameter(index=i, value={1: 1, 2: 2.0000, 3: 3.1415}, name='a')
>>> a
a[1]=1
a[2]=2.0
a[3]=3.1415
>>> round(a[i])
round(a[i])[1]=1
round(a[i])[2]=2
round(a[i])[3]=3
>>> round(a[i], 0)
round(a[i], 0)[1]=1
round(a[i], 0)[2]=2.0
round(a[i], 0)[3]=3.0
>>> round(a[i], 2)
round(a[i], 2)[1]=1
round(a[i], 2)[2]=2.0
round(a[i], 2)[3]=3.14
```

__setitem__(*key: ObjPrm | tuple[ObjPrm, ...] | Index, value: ObjPrm*) → None

key に対応した値を変更します．

パラメータ

- **key** (*ObjPrm* or tuple of them or Index) --
- **value** (*ObjPrm*) --

例外 **KeyError** -- key がキーに存在しない場合

サンプル

```
>>> i = Element(value=[1,2], name='i')
>>> j = Element(value=[3,4], name='j')
>>> a = Parameter(index=(i,j), name='a')
>>> a[i,j] = 1
>>> a
a[1,3]=1
a[1,4]=1
a[2,3]=1
a[2,4]=1
>>> a[i,3] = i*10
>>> a
a[1,3]=10
a[1,4]=1
a[2,3]=20
a[2,4]=1
>>> a[2,3] = 100
>>> a
a[1,3]=10
a[1,4]=1
a[2,3]=100
a[2,4]=1
>>> ij = Element(value=[(1,3), (1,4), (2,3)], name='ij')
>>> a[ij] = 0
>>> a
a[1,3]=0
a[1,4]=0
a[2,3]=0
a[2,4]=1
>>> a[1,5] = 1
KeyError: 'a[1,5]'
>>> b = Parameter(index=j, value={3: 4, 4: 3}, name='b')
>>> a[i, b[j]] = i+j
>>> a
a[1,3]=5
a[1,4]=4
a[2,3]=6
a[2,4]=5
```

get(**key: ObjPrm*) → DType | *Parameter*

key に対応した添字の Parameter を返します．__getitem__ と異なり key に対応するものがない場合には KeyError が投げられず 0 となります．key の次元が異なる場合はその限りではありません．デフォルト値の指定はできず常に 0 となります．

パラメータ ***key** (*ObjPrm*) --

戻り値 key がすべて *DType* の場合，*DType* を返します．それ以外の場合，Parameter を返します．

戻り値の型 *DType* or *Parameter*

例外 **KeyError** -- '{}' takes {} dimension index ({} given)

サンプル

```
>>> i = Element(value=[1, 2], name='i')
>>> j = Element(value=[3, 4], name='j')
>>> a = Parameter(index=(i,j), name='a'); a[i,j] = i+j
>>> a.get(i,j)
a.get(i,j)[1,3]=4
a.get(i,j)[1,4]=5
a.get(i,j)[2,3]=5
a.get(i,j)[2,4]=6
>>> a.get(i,4)
a.get(i,4)[1,4]=5
a.get(i,4)[2,4]=6
>>> a.get(i,5)
a.get(i,5)[1,5]=0
a.get(i,5)[2,5]=0
>>> a.get(i+1,j)
a.get((i+1)[i],j)[1,3]=5
a.get((i+1)[i],j)[1,4]=6
a.get((i+1)[i],j)[2,3]=0
a.get((i+1)[i],j)[2,4]=0
```

参考:

pysimple.Parameter.__getitem__

property index: *pysimple.index.Index*

添字

items() → dict_items[Key, VT]

キーと値ペアの列を返します．

keys() → dict_keys[Key, VT]

キーの列を返します．

property name: str

name 属性です．値は文字列の代入により変更できます．

values() → dict_values[Key, VT]

値の列を返します．

バージョン 1.3.0 で追加: *__round__()* が使用可能になりました.

バージョン 1.5.0 で追加: *get()* が添字をサポートするようになりました.

class pysimple.table.**Table**

.val などの属性を表すクラスです.

Table クラスは *__getitem__()*, *get()*, *__setitem__()* がないこと以外は，ほぼ *Parameter* と同じです.

Table クラスのコンストラクタは公開されません.

init, *val*, *lb*, *ub*, *dual*, *init*, *val*, *dual*, *violation* が Table クラスに該当します.

5.1.4 変数，式

class pysimple.**Variable**(*name: str = None*, *index: Set | ELEMENT | tuple[Set | ELEMENT, ...] | Index | None = None*, *type: Literal[float, int, bin] | None = float*, *init: float | dict | None = 0*, *lb: float | dict | None = float('-inf')*, *ub: float | dict | None = float('inf')*)

ベースクラス: pysimple.util.Final, collections.abc.Collection[tuple[int | float | str, ...]], pysimple.util.Named

変数を表すクラスです.

パラメータ

- **name** (*str*) -- 省略した場合は自動で与えられます．オブジェクトの生成後に変更できます.
- **index** (Set or *ELEMENT* or tuple of them or Index) -- Variable の添字．省略した場合，添字なしの変数となります.
- **type** (*float or int or bin*) -- 組み込み関数の float, int, bin を指定します．それぞれ連続変数，整数変数，0-1 整数変数を意味します．bin が指定された場合，自動で lb=0, ub=1 が設定されます．オブジェクトの生成後に変更できます.
- **init** (*float or dict*) -- Variable の初期値．float の場合，すべての要素に対して同じ値が適用されます．dict の場合，指定された要素の値が適用されます．オブジェクトの生成後に変更できます.
- **lb** (*float or dict*) -- Variable の下限値.
- **ub** (*float or dict*) -- Variable の上限値.

例外

- **TypeError** -- 'type' keyword only takes built-in function 'float', 'int' or 'bin'
- ***SimpleError*** -- infeasible bound of variable {} (defined infeasible bound [{} <= * <= {}])

サンプル

```
>>> i = Element(value=[1,2], name='i')
>>> x = Variable(index=i, name='x')
>>> x
x:
x[1]
x[2]
>>> x.init
x[1].init=0
x[2].init=0
>>> x.val
x[1].val=0
x[2].val=0
>>> y = Variable(index=i, init=1, name='y')
>>> y.val
y[1].val=1
y[2].val=1
>>> z = Variable(index=i, init={2: 2}, name='z')
>>> z.val
z[1].val=0
z[2].val=2
>>> j = Element(value=[3,4], name='j')
>>> w = Variable(index=(i,j), name='w')
>>> w.val
w[1,3].val=0
w[1,4].val=0
w[2,3].val=0
w[2,4].val=0
>>> u = Variable(name='u')
>>> u.val
u.val=0
```

参考:

pysimple.Variable.type, *pysimple.Variable.init*, *pysimple.Variable.val*, *pysimple.Variable.lb*, *pysimple.Variable.ub*

__contains__(*key: DType or Key*) → bool

key がキーに含まれているかを返します.

戻り値の型 bool

__getitem__(*key: ObjPrm | tuple[ObjPrm, ...] | Index*) → *Variable*

key に対応した添字の Variable を返します.

パラメータ **key** (*ObjPrm* or tuple of them or Index) --

戻り値の型 *Variable*

例外 **KeyError** -- key がキーに存在しない場合

サンプル

```
>>> i = Element(value=[1, 2], name='i')
>>> j = Element(value=[3, 4], name='j')
>>> x = Variable(index=(i,j), name='x'); x[i,j] = i+j
>>> x[i,j].val
x[1,3].val=4
x[1,4].val=5
x[2,3].val=5
x[2,4].val=6
>>> x[i,3].val
x[1,3].val=4
x[2,3].val=5
>>> x[2,3].val
5
>>> ij = Element(value=[(1,3), (1,4), (2,3)], name='ij')
>>> x[ij].val
x[1,3].val=4
x[1,4].val=5
x[2,3].val=5
>>> x[1,5]
KeyError: 'x[1,5]'
>>> b = Parameter(index=j, value={3: 4, 4: 3}, name='b')
>>> x[i, b[j]].val
x[i,b[j]][1,3].val=5
x[i,b[j]][1,4].val=4
x[i,b[j]][2,3].val=6
x[i,b[j]][2,4].val=5
```

参考:

pysimple.Variable.get

__iter__() → collections.abc.Iterator[Key]

iter(self) を返します.

戻り値の型 Iterator[*Key*]

サンプル

```
>>> i = Element(value=[1,2])
>>> x = Variable(index=i)
>>> for _i in x:
...     print(_i)
...
(1,)
(2,)
>>> ij = Element(value=[(1,3), (1,4), (2,3)])
>>> y = Variable(index=ij)
>>> for _ij in y:
...     print(_ij)
...
(1, 3)
(1, 4)
(2, 3)
```

注釈: 次元に関わらず，値は常にタプルとして返されることに注意してください.

__len__() → int

キーの要素数を返します.

戻り値の型 int

__setitem__(*key: ObjPrm | tuple[ObjPrm, ...] | Index*, *value: ObjPrm*) → None

key に対応した値を変更します．値は val で確認できます.

パラメータ

- **key** (*ObjPrm* or tuple of them or Index) --
- **value** (*ObjPrm*) --

例外 **KeyError** -- key がキーに存在しない場合

サンプル

```
>>> i = Element(value=[1,2], name='i')
>>> j = Element(value=[3,4], name='j')
>>> x = Variable(index=(i,j), name='x')
>>> x[i,j] = 1
>>> x.val
x[1,3].val=1
x[1,4].val=1
```

(次のページに続く)

(前のページからの続き)

```
x[2,3].val=1
x[2,4].val=1
>>> x[i,3] = i*10
>>> x.val
x[1,3].val=10
x[1,4].val=1
x[2,3].val=20
x[2,4].val=1
>>> x[2,3] = 100
>>> x.val
x[1,3].val=10
x[1,4].val=1
x[2,3].val=100
x[2,4].val=1
>>> ij = Element(value=[(1,3), (1,4), (2,3)], name='ij')
>>> x[ij] = 0
>>> x.val
x[1,3].val=0
x[1,4].val=0
x[2,3].val=0
x[2,4].val=1
>>> x[1,5] = 1
KeyError: 'x[1,5]'
>>> b = Parameter(index=j, value={3: 4, 4: 3}, name='b')
>>> x[i, b[j]] =  i+j
>>> x.val
x[1,3].val=5
x[1,4].val=4
x[2,3].val=6
x[2,4].val=5
```

property dual: *pysimple.table.Table* | None

変数の双対変数値を表す属性です．値は求解時に更新されます．オブジェクト生成時の値は None です．

fix() → None

変数の値を現在値 val に固定します．下限値 lb と上限値 ub を現在値 val に設定するのと同じ効果です．lb/ub と値が矛盾する場合は求解時に SimpleError が投げられます．

参考:

pysimple.Variable.lb, *pysimple.Variable.ub*, *pysimple.Variable.unfix*

get(**key: ObjPrm*) → *Variable*

key に対応した添字の Variable を返します．__getitem__ と異なり key に対応するものがない場合には KeyError が投げられず 0 となります．key の次元が異なる場合はその限りではありません．デフォルト値の指定はできず常に 0 となります．

パラメータ ***key** (*ObjPrm*) --

戻り値の型 *Variable*

例外 **KeyError** -- '{}' takes {} dimension index ({} given)

サンプル

```
>>> i = Element(value=[1, 2], name='i')
>>> j = Element(value=[3, 4], name='j')
>>> x = Variable(index=(i,j), name='x'); x[i,j] = i+j
>>> x.get(i,j)
x.get(i,j):
x.get(i,j)[1,3]
x.get(i,j)[1,4]
x.get(i,j)[2,3]
x.get(i,j)[2,4]
>>> x.get(i,4)
x.get(i,4):
x.get(i,4)[1,4]
x.get(i,4)[2,4]
>>> x.get(i,5)
x.get(i,5):
0
0
>>> x.get(i+1,j)
x.get((i+1)[i],j):
x.get((i+1)[i],j)[1,3]
x.get((i+1)[i],j)[1,4]
0
0
```

参考:

pysimple.Variable.__getitem__

property group: **tuple[int, ...] | None**

変数の添字グループ情報を表す属性です． グループは添字の次元の列で指定します．解法が wls である場合，添字グループは wls における割当ラベルとして解釈され，解の探索に利用されます．具体的には，アルゴリズム内で各変数が「指定された次元の列」と「指定されなかった次元の列」を割り当てる（関連づける）構造をもつと解釈されます．例えば，下記サンプルコードの変数 x が「i を (j, k) に割り当てるなら 1，そうでないなら 0」を意味する変数である場合，x.group = 0 の設定により探索性能の向上が見込めます．本属性は解法 wls でのみ有効です．

サンプル

```
>>> i = Element(value=[1,2], name='i')
>>> j = Element(value=[3,4], name='j')
>>> k = Element(value=[5,6], name='k')
>>> x = BinaryVariable(index=(i,j,k), name='x')
>>> x.group = 0
>>> x.group
(0,)
>>> x.group = 0, 2
>>> x.group
(0, 2)
```

property index: *pysimple.index.Index*

添字

property init: *pysimple.table.Table*

変数の初期値を表す属性です．値は初期化，値の代入，求解時に更新されます．求解時には .val 属性の値からコピーされます．

サンプル

```
>>> p = Problem(silent=True)
>>> i = Element(value=[1,2])
>>> x = Variable(index=i, init=2, lb=1, name='x')
>>> x.init
x[1].init=2
x[2].init=2
>>> x[2] = 3
>>> x.init
x[1].init=2
x[2].init=3
>>> p += Sum(x[i])
>>> p.solve()
<NuoptStatus.OPTIMAL: 1>
>>> x.init
x[1].init=2
x[2].init=3
>>> p.solve()
<NuoptStatus.OPTIMAL: 1>
>>> x.init
x[1].init=1.0000000002083331
x[2].init=1.0000000002083331
```

参考:

pysimple.Variable.val

property lb: *pysimple.table.Table*

変数の下限値を表す属性です．値は初期化時にのみ更新されます．初期化時に type=bin を指定すると lb=0 が設定されます．

サンプル

```
>>> i = Element(value=[1,2], name='i')
>>> x = Variable(index=i, name='x')
>>> x.lb
x[1].lb=-inf
x[2].lb=-inf
>>> y = Variable(index=i, lb=0, name='y')
>>> y.lb
y[1].lb=0
y[2].lb=0
>>> z = BinaryVariable(index=i, name='z')
>>> z.lb
z[1].lb=0
z[2].lb=0
```

参考:

pysimple.Variable.type, *pysimple.Variable.ub*

property name: str

name 属性です．値は文字列の代入により変更できます．

property type: Literal[float | int | bin]

変数の種類を表す属性です．値は代入により変更できます．組み込み関数の float, int, bin を指定します．それぞれ連続変数，整数変数，0-1 整数変数を意味します．

サンプル

```
>>> x = Variable()
>>> x.type
<class 'float'>
>>> x.type = int
>>> x.type
<class 'int'>
>>> x.type = bin
>>> x.type
```

(次のページに続く)

(前のページからの続き)

```
<built-in function bin>
>>> x.type = bool
TypeError: 'type' keyword only takes built-in function 'float', 'int' or 'bin
↪'
>>> y1 = Variable(type=int)
>>> y1.type
<class 'int'>
>>> y2 = IntegerVariable()
>>> y2.type
<class 'int'>
```

参考:

pysimple.Variable.lb, *pysimple.Variable.ub*, *pysimple.IntegerVariable*, *pysimple.BinaryVariable*

property ub: *pysimple.table.Table*

変数の上限値を表す属性です．値は初期化時にのみ更新されます．初期化時に type=bin を指定すると ub=1 が設定されます．

サンプル

```
>>> i = Element(value=[1,2], name='i')
>>> x = Variable(index=i, name='x')
>>> x.ub
x[1].ub=inf
x[2].ub=inf
>>> y = Variable(index=i, ub=1, name='y')
>>> y.ub
y[1].ub=1
y[2].ub=1
>>> z = BinaryVariable(index=i, name='z')
>>> z.ub
z[1].ub=1
z[2].ub=1
```

参考:

pysimple.Variable.type, *pysimple.Variable.lb*

unfix() → None

Variable.fix で固定した変数の値を解除します．

参考:

pysimple.Variable.fix

property val: *pysimple.table.Table*

変数の現在値を表す属性です．値は初期化，値の代入，求解時に更新されます．

サンプル

```
>>> p = Problem(silent=True)
>>> i = Element(value=[1,2])
>>> x = Variable(index=i, init=2, lb=1, name='x')
>>> x.val
x[1].val=2
x[2].val=2
>>> x[2].val
2
>>> x[2] = 3
>>> x.val
x[1].val=2
x[2].val=3
>>> p += Sum(x[i])
>>> p.solve()
<NuoptStatus.OPTIMAL: 1>
>>> x.val
x[1].val=1.0000000002083331
x[2].val=1.0000000002083331
```

参考:

pysimple.Variable.init

バージョン 1.5.0 で追加: *fix()* , *unfix()* が追加されました．

バージョン 1.5.0 で追加: *get()* が追加されました．

class pysimple.**IntegerVariable**(*name: str = None*, *index:* Set | *ELEMENT* | *tuple[*Set | *ELEMENT, ...]* | Index | *None = None*, *type: Literal[float, int, bin]* | *None = float*, *init: float* | *dict* | *None = 0*, *lb: float* | *dict* | *None = float('-inf')*, *ub: float* | *dict* | *None = float('inf')*)

ベースクラス: *pysimple.expression.Variable*

整数変数を表すクラスです．Variable(type=int, ..) と等価です．type キーワードはありません．

パラメータ

- **name** (*str*) -- 省略した場合は自動で与えられます．オブジェクトの生成後に変更できます．
- **index** (Set or *ELEMENT* or tuple of them or Index) -- Variable の添字．省略した場合，添字なしの変数となります．

- **type** (*float or int or bin*) -- 組み込み関数の float, int, bin を指定します．それぞれ連続変数，整数変数，0-1 整数変数を意味します．bin が指定された場合，自動で lb=0, ub=1 が設定されます．オブジェクトの生成後に変更できます．
- **init** (*float or dict*) -- Variable の初期値．float の場合，すべての要素に対して同じ値が適用されます．dict の場合，指定された要素の値が適用されます．オブジェクトの生成後に変更できます．
- **lb** (*float or dict*) -- Variable の下限値．
- **ub** (*float or dict*) -- Variable の上限値．

例外

- **TypeError** -- 'type' keyword only takes built-in function 'float', 'int' or 'bin'
- ***SimpleError*** -- infeasible bound of variable {} (defined infeasible bound [{} <= * <= {}])

サンプル

```
>>> i = Element(value=[1,2], name='i')
>>> x = Variable(index=i, name='x')
>>> x
x:
x[1]
x[2]
>>> x.init
x[1].init=0
x[2].init=0
>>> x.val
x[1].val=0
x[2].val=0
>>> y = Variable(index=i, init=1, name='y')
>>> y.val
y[1].val=1
y[2].val=1
>>> z = Variable(index=i, init={2: 2}, name='z')
>>> z.val
z[1].val=0
z[2].val=2
>>> j = Element(value=[3,4], name='j')
>>> w = Variable(index=(i,j), name='w')
>>> w.val
w[1,3].val=0
w[1,4].val=0
w[2,3].val=0
w[2,4].val=0
```

(次のページに続く)

(前のページからの続き)

```
>>> u = Variable(name='u')
>>> u.val
u.val=0
```

参考:

pysimple.Variable.type, *pysimple.Variable.init*, *pysimple.Variable.val*, *pysimple.Variable.lb*, *pysimple.Variable.ub*

class pysimple.**BinaryVariable**(*name: str = None*, *index: Set | ELEMENT | tuple[Set | ELEMENT, ...] | Index | None = None*, *type: Literal[float, int, bin] | None = float*, *init: float | dict | None = 0*, *lb: float | dict | None = float('-inf')*, *ub: float | dict | None = float('inf')*)

ベースクラス: *pysimple.expression.Variable*

0-1 整数変数を表すクラスです．Variable(type=bin, ..) と等価です．type キーワードはありません．

パラメータ

- **name** (*str*) -- 省略した場合は自動で与えられます．オブジェクトの生成後に変更できます．
- **index** (Set or *ELEMENT* or tuple of them or Index) -- Variable の添字．省略した場合，添字なしの変数となります．
- **type** (*float or int or bin*) -- 組み込み関数の float, int, bin を指定します．それぞれ連続変数，整数変数，0-1 整数変数を意味します．bin が指定された場合，自動で lb=0, ub=1 が設定されます．オブジェクトの生成後に変更できます．
- **init** (*float or dict*) -- Variable の初期値．float の場合，すべての要素に対して同じ値が適用されます．dict の場合，指定された要素の値が適用されます．オブジェクトの生成後に変更できます．
- **lb** (*float or dict*) -- Variable の下限値．
- **ub** (*float or dict*) -- Variable の上限値．

例外

- **TypeError** -- 'type' keyword only takes built-in function 'float', 'int' or 'bin'
- ***SimpleError*** -- infeasible bound of variable {} (defined infeasible bound [{} <= * <= {}])

サンプル

```
>>> i = Element(value=[1,2], name='i')
>>> x = Variable(index=i, name='x')
>>> x
x:
x[1]
x[2]
>>> x.init
x[1].init=0
x[2].init=0
>>> x.val
x[1].val=0
x[2].val=0
>>> y = Variable(index=i, init=1, name='y')
>>> y.val
y[1].val=1
y[2].val=1
>>> z = Variable(index=i, init={2: 2}, name='z')
>>> z.val
z[1].val=0
z[2].val=2
>>> j = Element(value=[3,4], name='j')
>>> w = Variable(index=(i,j), name='w')
>>> w.val
w[1,3].val=0
w[1,4].val=0
w[2,3].val=0
w[2,4].val=0
>>> u = Variable(name='u')
>>> u.val
u.val=0
```

参考:

pysimple.Variable.type, *pysimple.Variable.init*, *pysimple.Variable.val*, *pysimple.Variable.lb*, *pysimple.Variable.ub*

class pysimple.expression.**QVariable**

ベースクラス: *pysimple.expression.Variable*

二次の変数を表すクラスです.

QVariable クラスのコンストラクタは公開されません.

class pysimple.expression.**Expression**

ベースクラス: collections.abc.Collection[tuple[int | float | str, ...]], pysimple.util.Named

式を表すクラスです.

__contains__(*key: DType or Key*) → bool

key がキーに含まれているかを返します.

戻り値の型 bool

__getitem__(*key: ObjPrm | tuple[ObjPrm, ...] | Index*) → *Expression*

key に対応した添字の Expression を返します.

パラメータ **key** (*ObjPrm* or tuple of them or Index) --

戻り値の型 *Expression*

例外 **KeyError** -- key がキーに存在しない場合

サンプル

```
>>> i = Element(value=[1, 2], name='i')
>>> j = Element(value=[3, 4], name='j')
>>> x = Variable(index=(i,j), name='x'); x[i,j] = i+j
>>> e = x[i,j] + 1
>>> e[i,j].val
(x[i,j]+1)[1,3].val=5
(x[i,j]+1)[1,4].val=6
(x[i,j]+1)[2,3].val=6
(x[i,j]+1)[2,4].val=7
>>> e[i,3].val
(x[i,j]+1)[1,3].val=5
(x[i,j]+1)[2,3].val=6
>>> e[2,3].val
6
>>> ij = Element(value=[(1,3), (1,4), (2,3)], name='ij')
>>> e[ij].val
(x[i,j]+1)[1,3].val=5
(x[i,j]+1)[1,4].val=6
(x[i,j]+1)[2,3].val=6
>>> e[1,5].val
KeyError: 'x[1,5]'
>>> b = Parameter(index=j, value={3: 4, 4: 3}, name='b')
>>> e[i, b[j]].val
(x[i,j]+1)[i,b[j]][1,3].val=6
(x[i,j]+1)[i,b[j]][1,4].val=5
(x[i,j]+1)[i,b[j]][2,3].val=7
(x[i,j]+1)[i,b[j]][2,4].val=6
```

参考:

pysimple.expression.Expression.get

__iter__() → collections.abc.Iterator[Key]

iter(self) を返します.

戻り値の型 Iterator[*Key*]

サンプル

```
>>> i = Element(value=[1,2])
>>> x = Variable(index=i)
>>> x1 = x[i] + 1
>>> for _i in x1:
...     print(_i)
...
(1,)
(2,)
>>> ij = Element(value=[(1,3), (1,4), (2,3)])
>>> y = Variable(index=ij)
>>> y1 = y[ij] + 1
>>> for _ij in y1:
...     print(_ij)
...
(1, 3)
(1, 4)
(2, 3)
```

注釈: 次元に関わらず，値は常にタプルとして返されることに注意してください.

__len__()

キーの要素数を返します.

戻り値の型 int

get(**key: ObjPrm*) → *Expression*

key に対応した添字の Expression を返します．__getitem__ と異なり key に対応するものがない場合には KeyError が投げられず 0 となります．key の次元が異なる場合はその限りではありません．デフォルト値の指定はできず常に 0 となります.

パラメータ ***key** (*ObjPrm*) --

戻り値の型 *Expression*

例外 **KeyError** -- '{}' takes {} dimension index ({} given)

サンプル

```
>>> i = Element(value=[1, 2], name='i')
>>> j = Element(value=[3, 4], name='j')
>>> x = Variable(index=(i,j), name='x'); x[i,j] = i+j
>>> e = x[i,j] + 1
>>> e.get(i,j)
(x[i,j]+1).get(i,j):
x[1,3]+1
x[1,4]+1
x[2,3]+1
x[2,4]+1
>>> e.get(i,4)
(x[i,j]+1).get(i,4):
x[1,4]+1
x[2,4]+1
>>> e.get(i,5)
(x[i,j]+1).get(i,5):
1
1
>>> e.get(i+1,j)
(x[i,j]+1).get((i+1)[i],j):
x[2,3]+1
x[2,4]+1
1
1
```

参考:

pysimple.expression.Expression.__getitem__

property index: *pysimple.index.Index*

添字

property init: *pysimple.table.Table*

式の初期値を表す属性です．値は構成される変数・定数の値の更新に連動します．

サンプル

```
>>> p = Problem(silent=True)
>>> i = Element(value=[1,2], name='i')
>>> x = Variable(index=i, init=2, lb=1, name='x')
>>> x1 = x[i] + 1
>>> x1.init
(x[i]+1)[1].init=3
(x[i]+1)[2].init=3
```

(次のページに続く)

(前のページからの続き)

```
>>> x[2] = 3
>>> x1.init
(x[i]+1)[1].init=3
(x[i]+1)[2].init=4
>>> p += Sum(x1[i])
>>> p.solve()
<NuoptStatus.OPTIMAL: 1>
>>> x1.init
(x[i]+1)[1].init=3
(x[i]+1)[2].init=4
>>> p.solve()
<NuoptStatus.OPTIMAL: 1>
>>> x1.init
(x[i]+1)[1].init=2.000000000208333
(x[i]+1)[2].init=2.000000000208333
```

参考:

pysimple.expression.Expression.val

property name: str

name 属性です．値は文字列の代入により変更できます．

property val: *pysimple.table.Table*

式の現在値を表す属性です．値は構成される変数の値の更新に連動します．

サンプル

```
>>> p = Problem(silent=True)
>>> i = Element(value=[1,2], name='i')
>>> x = Variable(index=i, init=2, lb=1, name='x')
>>> x1 = x[i] + 1
>>> x1.val
(x[i]+1)[1].val=3
(x[i]+1)[2].val=3
>>> x[2] = 3
>>> x1.val
(x[i]+1)[1].val=3
(x[i]+1)[2].val=4
>>> p += Sum(x1[i])
>>> p.solve()
<NuoptStatus.OPTIMAL: 1>
>>> x1.val
(x[i]+1)[1].val=2.000000000208333
(x[i]+1)[2].val=2.000000000208333
```

参考:

pysimple.expression.Expression.init

バージョン 1.3.1 で追加: *__getitem__()* に Parameter が使用可能になりました.

バージョン 1.5.0 で追加: *get()* が追加されました.

5.1.5 制約式

class pysimple.constraint.**Constraint**

ベースクラス: collections.abc.Collection[tuple[int | float | str, ...]], pysimple.util.Named

制約式を表すクラスです.

Constraint クラスのコンストラクタは公開されません.

__contains__(*key: DType or Key*) → bool

key がキーに含まれているかを返します.

戻り値の型 bool

__getitem__(*key: ObjPrm | tuple[ObjPrm, ...] | Index*) → *Constraint*

key に対応した添字の Constraint を返します.

パラメータ **key** (*ObjPrm* or tuple of them or Index) --

戻り値の型 *Constraint*

例外 **KeyError** -- key がキーに存在しない場合

サンプル

```
>>> i = Element(value=[1, 2], name='i')
>>> j = Element(value=[3, 4], name='j')
>>> x = Variable(index=(i,j), name='x'); x[i,j] = i+j
>>> cons = x[i,j] >= 1
>>> cons[i,j]
(x[i,j]>=1):
x[1,3]>=1
x[1,4]>=1
x[2,3]>=1
x[2,4]>=1
>>> cons[i,3]
(x[i,j]>=1):
x[1,3]>=1
x[2,3]>=1
>>> cons[2,3]
```

(次のページに続く)

(前のページからの続き)

```
(x[i,j]>=1):
x[2,3]>=1
>>> ij = Element(value=[(1,3), (1,4), (2,3)], name='ij')
>>> cons[ij]
(x[i,j]>=1):
x[1,3]>=1
x[1,4]>=1
x[2,3]>=1
>>> cons[1,5]
KeyError: 'x[1,5]'
```

参考:

pysimple.constraint.Constraint.get

__iter__() → collections.abc.Iterator[Key]

iter(self) を返します.

戻り値の型 Iterator[*Key*]

サンプル

```
>>> i = Element(value=[1,2])
>>> x = Variable(index=i)
>>> cons1 = x[i] <= 1
>>> for _i in cons1:
...     print(_i)
...
(1,)
(2,)
>>> ij = Element(value=[(1,3), (1,4), (2,3)])
>>> y = Variable(index=ij)
>>> cons2 = y[ij] <= 1
>>> for _ij in cons2:
...     print(_ij)
...
(1, 3)
(1, 4)
(2, 3)
```

注釈: 次元に関わらず，値は常にタプルとして返されることに注意してください.

__len__() → int

キーの要素数を返します.

戻り値の型 int

property dual: *pysimple.table.Table* | None

制約式の双対変数値を表す属性です．値は求解時に更新されます．オブジェクト生成時の値は None です．

get(**key: ObjPrm*) → *Constraint*

key に対応した添字の Constraint を返します．__getitem__ と異なり key に対応するものがない場合には KeyError が投げられず 0 となります．key の次元が異なる場合はその限りではありません．デフォルト値の指定はできず常に 0 となります．

パラメータ ***key** (*ObjPrm*) --

戻り値の型 *Constraint*

例外 **KeyError** -- '{}' takes {} dimension index ({} given)

サンプル

```
>>> i = Element(value=[1, 2], name='i')
>>> j = Element(value=[3, 4], name='j')
>>> x = Variable(index=(i,j), name='x'); x[i,j] = i+j
>>> cons = x[i,j] >= 1
>>> cons.get(i,j)
(x[i,j]>=1):
x[1,3]>=1
x[1,4]>=1
x[2,3]>=1
x[2,4]>=1
>>> cons.get(i,4)
(x[i,j]>=1):
x[1,4]>=1
x[2,4]>=1
>>> cons.get(i,5)
(x[i,j]>=1):
0>=1
0>=1
>>> cons.get(i+1,j)
(x[i,j]>=1):
x[2,3]>=1
x[2,4]>=1
0>=1
0>=1
```

参考:

pysimple.constraint.Constraint.__getitem__

property index: *pysimple.index.Index*

添字

isHard() → bool

weight がハード制約の場合に True となります．*SelectionConstraint* では常に True です．

isSemiHard() → bool

weight がセミハード制約の場合に True となります．

isSoft() → bool

weight がソフト制約の場合に True となります．

property name: str

name 属性です．値は文字列の代入により変更できます．

property violation: *pysimple.constraint.ConstraintWeight*

制約式の違反値を表す属性です．両辺の現在値に対して計算されます．不等式制約と等式制約の場合で異なり，以下が成立します．

- (式 1 <= 式 2).violation = 式 1.val - 式 2.val
- (式 1 >= 式 2).violation = 式 2.val - 式 1.val
- (式 1 == 式 2).violation = abs(式 1.val - 式 2.val)

値は構成される変数・定数の値の更新に連動します．

サンプル

```
>>> i = Element(value=[1,2], name='i')
>>> x = Variable(index=i, init=2, name='x')
>>> (x[i] <= 5).violation
(x[i]<=5)[1].violation=-3
(x[i]<=5)[2].violation=-3
>>> y = Variable(index=i, init=9, name='y')
>>> (y[i] <= 5).violation
(y[i]<=5)[1].violation=4
(y[i]<=5)[2].violation=4
>>> z = Variable(index=i, init=2, name='z')
>>> (z[i] == 5).violation
(z[i]==5)[1].violation=3
(z[i]==5)[2].violation=3
>>> w = Variable(index=i, init=9, name='w')
>>> (w[i] == 5).violation
(w[i]==5)[1].violation=4
(w[i]==5)[2].violation=4
```

property weight: *pysimple.constraint.ConstraintWeight*

制約関数で指定した制約種別です．

バージョン 1.5.0 で追加: *get()* が追加されました．

class pysimple.constraint.**SelectionConstraint**

ベースクラス: *pysimple.constraint.Constraint*

Selection 関数によって作成された制約を表すクラスです．

class pysimple.constraint.**ConstraintWeight**

制約式の種別を表すクラスです．

参考:

HardConstraint, *SemiHardConstraint*, *SoftConstraint*

5.1.6 問題，求解オプション，解情報

class pysimple.**Problem**(*name: str = None*, *type: Literal[min, max] = min*, *silent: bool = False*, *subprocess: bool = False*, *solfile: bool | str | None = None*, *iis: bool = True*)

問題を表すクラスです．

パラメータ

- **name** (*str*) -- 省略した場合はクラス名になります．オブジェクトの生成後に変更できます．
- **type** (*min or max*) -- 組み込み関数の min, max を指定します．それぞれ目的関数の最小化，最大化を意味します．
- **silent** (*bool*) -- 求解時の求解情報を標準出力に出力するか制御します．False: 出力する, True: 出力しない
- **subprocess** (*bool*) -- 求解を別プロセスとして行うかを制御します．False: 同一プロセスで行う, True: 別プロセスで行う
- **solfile** (*None or bool or str*) -- 解ファイルの出力を制御します．
  - None or False: 出力しない．
  - True: 出力する．ファイル名は Problem.name + '.sol' となります．
  - str: 出力する．ファイル名は solfile + '.sol' となります．
- **iis** (*bool*) -- 解が infeasible の場合に IIS 情報を生成するかどうか制御します. False: IIS 情報を生成しない, True: IIS 情報を生成する

注釈: 引数 solfile でファイル名を指定する場合は絶対パス名，相対パス名のいずれでも指定することができます．また，以下の場合に solfile の出力に失敗しますが，Variable.val 等で計算結果の取得は可能です．

- ファイル名として使用できない文字 (: * ? " < > |) が含まれている場合

- 同名のファイルがすでに存在し，上書きの許可がない場合
- ファイルシステムに空き容量がない場合

参考:

pysimple.Problem.solve

__delitem__(*key: str | int*) → None

del problem.constraints[key] と等価です.

パラメータ **key** (*str or int*) -- 削除する制約式名，または追加した制約式の番号．負数の場合，逆順から参照されます.

サンプル

```
>>> p = Problem()
>>> x = Variable(name='x')
>>> y = Variable(name='y')
>>> p += x >= 1
>>> p += y >= 2, ' 制約式 2'
>>> p
Problem(name='Problem', type=min, silent=False, subprocess=False,␣
↪solfile=None, iis=True):
[constraints]
(x>=1):
x>=1
制約式 2:
y>=2
.
[objective]
None
.
>>> del p['(x>=1)']
>>> del p[' 制約式 2']
>>> p
Problem(name='Problem', type=min, silent=False, subprocess=False,␣
↪solfile=None, iis=True):
[constraints]
.
[objective]
None
```

__getitem__(*key: str | int*) → *Constraint*

problem.constraints[key] と等価です.

パラメータ **key** (*str or int*) -- 参照する制約式名，または追加した制約式の番号．負数の場合，逆順から参照されます．

サンプル

```
>>> p = Problem()
>>> x = Variable(name='x')
>>> y = Variable(name='y')
>>> p += x >= 1
>>> p += y >= 2, ' 制約式 2'
>>> p
Problem(name='Problem', type=min, silent=False, subprocess=False,␣
↪solfile=None, iis=True):
[constraints]
(x>=1):
x>=1
制約式 2:
y>=2
.
[objective]
None
.
>>> p['(x>=1)']
(x>=1):
x>=1
>>> p[' 制約式 2']
制約式 2:
y>=2
>>> p[1]
制約式 2:
y>=2
>>> p[-1]
制約式 2:
y>=2
```

__iadd__(*args:* Variable | Expression) → Self

__iadd__(*args:* *tuple*[Variable | Expression, *str*]) → Self

__iadd__(*args:* *tuple*[Variable | Expression, ConstraintWeight]) → Self

__iadd__(*args:* *tuple*[Variable | Expression, *str*, ConstraintWeight]) → Self

__iadd__(*args:* *tuple*[Variable | Expression, ConstraintWeight, *str*]) → Self

__iadd__(*args:* Constraint) → Self

__iadd__(*args:* *tuple*[Constraint, *str*]) → Self

問題に目的関数・制約式を設定します．「問題 += 変数・式 [, 目的関数名]」で目的関数を設定し，「問題 += 制約式 [, 制約式名]」で制約式を追加します．目的関数には添字が残っていてはい

けません．目的関数を複数設定することは通常できませんが，一旦求解することで再設定できるようになります．同名の制約式を追加することはできません．

例外

- ***SimpleError*** -- objective can only be assigned once
- ***SimpleError*** -- override constraint {}

注釈: 目的関数名・制約式名を与えた場合は副作用があります．すなわち，式・制約式の name 属性が変更されます．

サンプル

```
>>> p = Problem(silent=True)
>>> i = Element(value=[1,2], name='i')
>>> x = Variable(index=i, name='x')
>>> p += Sum(x[i]), '総コスト'
>>> p += x[1] >= 1
>>> p += x[2] >= 2, '制約式 2'
>>> p
Problem(name='Problem', type=min, silent=True, subprocess=False,␣
↪solfile=None, iis=True):
[constraints]
(x[1]>=1):
x[1]>=1
制約式 2:
x[2]>=2
.
[objective]
総コスト:
x[1]+x[2]
.
>>> p += Sum(x[i])
pysimple.error.SimpleError: objective can only be assigned once
>>> p += x[1] >= 1
pysimple.error.SimpleError: override constraint '(x[1]>=1)'
>>> p.solve()
<NuoptStatus.OPTIMAL: 1>
>>> p += Sum(x[i])
```

参考:

pysimple.Problem.objective, *pysimple.Problem.constraints*

property constraints:
pysimple.problem.IndexedDict[*pysimple.constraint.Constraint*]

問題の制約式を表す属性です．「問題 += 制約式 [, 制約式名]」で追加されます．制約式名をキー，制約式を値とする辞書です．参照，削除には追加した順の番号も使用できます．負数の場合，逆順から参照されます．

サンプル

```
>>> x = Variable(name='x')
>>> y = Variable(name='y')
>>> p = Problem()
>>> p += x >= 1, 'cons1'
>>> p.constraints
cons1:
x>=1
>>> p += y >= 2, 'cons2'
>>> p.constraints
cons1:
x>=1
cons2:
y>=2
>>> p.constraints['cons2']
cons2:
y>=2
>>> p.constraints[1]
cons2:
y>=2
>>> p.constraints[-1]
cons2:
y>=2
>>> del p.constraints[-1]
>>> p.constraints
cons1:
x>=1
```

参考:

pysimple.Problem.__iadd__, *pysimple.Problem.objective*, *pysimple.Problem.variables*

hasSolution() → bool

直前の Problem.solve() で解情報が設定されているかどうかを返します．解情報は本問題で使用している変数の val 属性から参照することができます．

戻り値 求解後に解情報が設定されている場合に真，そうでない場合に偽を返します．

戻り値の型 bool

注釈: problem.status の値が NuoptStatus.OPTIMAL，NuoptStatus.FEASIBLE の場合は problem.hasSolution() は必ず真を返し，設定されている解情報は実行可能な解の保証があります．一方，problem.status がその他の値で problem.hasSolution() が真の場合は，実行不可能な解が設定されています．

参考:

pysimple.Problem.solve, *pysimple.Problem.status*, *pysimple.NuoptStatus*, *pysimple.Variable.val*

isFeasible() → bool

直前の Problem.solve() で実行可能解が求まったかどうかを返します．

戻り値 実行可能解が求まった場合に真を返します．実行可能解が求まっていない場合は偽を返します．

戻り値の型 bool

注釈: problem.status が NuoptStatus.OPTIMAL または NuoptStatus.FEASIBLE の場合に真を返します．本メソッドの戻り値が真の場合，本問題で使用している変数の val 属性から参照できる解情報は実行可能です．

参考:

pysimple.Problem.solve, *pysimple.Problem.status*, *pysimple.NuoptStatus*, *pysimple.Variable.val*

isInfeasible() → bool

直前の Problem.solve() で実行可能解が存在しないと判断されたかどうかを返します．

戻り値 求解後の実行可能解が存在しないと判断された場合に真，そうでない場合に偽を返します．

戻り値の型 bool

注釈: problem.status が NuoptStatus.INFEASIBLE の場合に真を返します．本メソッドの戻り値が真 (実行可能解が存在しない場合) で，かつ，problem.hasSolution() も真の場合は，本問題で使用している変数の val 属性には実行不可能な解情報が設定されています．

参考:

pysimple.Problem.solve, *pysimple.Problem.status*, *pysimple.Problem.hasSolution*, *pysimple.NuoptStatus*, *pysimple.Variable.val*

mpsout(*mpsfile: bool | str | None = True*, *anonymous: bool = False*, *initial: bool = False*) → None

問題情報を mps 形式でファイル出力します．

パラメータ

- **mpsfile** (*None or bool or str*) --
  - None or False: 出力しない.
  - True: 出力する．ファイル名は Problem.name + '.mps' となります.
  - str: 出力する．ファイル名は mpsfile + '.mps' となります.
- **anonymous** (*bool*) --
  - True: モデル情報を mps ファイルに埋め込まない.
  - False: モデル情報を mps ファイルに埋め込む.
- **initial** (*bool*) --
  - True: 初期値を出力します.
  - False: 初期値を出力しません.

注釈: 引数で指定された名前に拡張子 .mps が付与されたファイル名で mps 形式のファイルが出力されます．ファイル名は絶対パス名，相対パス名のいずれでも指定することができます．出力される mps ファイルの形式は自由長形式です（固定長形式ではありません）．以下の場合に mps ファイルの出力に失敗します.

- ファイル名として使用できない文字 (: * ? " < > |) が含まれている場合
- 同名のファイルがすでに存在し，上書きの許可がない場合
- ファイルシステムに空き容量がない場合

---

property name: *str*

name 属性です．値は文字列の代入により変更できます.

property objective: *pysimple.expression.Variable* | *pysimple.expression.Expression*

問題の目的関数を表す属性です．「問題 += 変数・式 [, 目的関数名]」で設定されます．オブジェクト生成時の値は空の式です．通常，目的関数を複数設定することはできませんが，一旦求解することで再設定できるようになります．del 文で目的関数を削除できます.

サンプル

```
>>> p = Problem(silent=True)
>>> i = Element(value=[1,2], name='i')
>>> x = Variable(index=i, lb=1, init=2, name='x')
>>> x.val
x[1].val=2
x[2].val=2
>>> p.objective
<empty>:
```

(次のページに続く)

(前のページからの続き)

```
0
>>> p += Sum(x[i])
>>> p.objective.val
4
>>> p += x[2]
pysimple.error.SimpleError: objective can only be assigned once
>>> p.solve()
<NuoptStatus.OPTIMAL: 1>
>>> x.val
x[1].val=1.0000000002083331
x[2].val=1.0000000002083331
>>> p.objective.val
2.0000000004166663
>>> p += x[2]
>>> del p.objective
>>> p += x[1]
```

参考:

pysimple.Problem.__iadd__, *pysimple.Problem.constraints*, *pysimple.Problem.variables*

property options: ***pysimple.options.ProblemOptions***

求解オプション定数を制御する属性です．種類と値は help(problem.options) か dir(problem.options) で，定数値は help(Options) か help(Options.Branch) で確認してください．

参考:

pysimple.options.ProblemOptions, *pysimple.Options*

property result: ***pysimple.problem.Result***

求解後の求解情報を表す属性です．詳細は help(problem.result) で確認してください．

参考:

pysimple.problem.Result

setCallback(*mip_terminate: Callable[[dict[str, int | float]], bool] | None = None*) → None

分枝限定法の停止を制御するコールバック関数を設定します．コールバック関数は次のような関数になります．

```
def mip_terminate(solver:  dict[str, int | float]) -> bool:
```

戻り値は bool 型オブジェクトが想定され，True を返すと分枝限定法を停止し，False を返すと継続して実行されます．引数には，本関数が呼び出された時点の最適化計算の情報が辞書型に格納されており，次の情報を取得することができます．

キー	内容
ElapsedTime	経過時間
RelativeGap	相対ギャップ
AbsoluteGap	絶対ギャップ
Objective	目的関数値
SolutionCount	実行可能解の数
TotalMemory	メモリ消費量 (MiB)
MemoryAvailable	利用可能メモリ量 (MiB)
PartialProblemCount	部分問題の数 (ノード数)

パラメータ **mip_terminate** (**コールバック関数** *or None*) -- 分枝限定法の終了コールバック関数を指定します．None を指定した場合はコールバック関数がクリアされます．

例外 **TypeError** -- mip_terminate が関数でも None でもない場合に raise されます．

注釈:

- **コールバック関数により停止した場合のステータス番号は次のようになります．**
  - 実行可能解が得られている場合 : NUOPT 31
  - 実行可能解が得られていない場合 : NUOPT 32
- 実行可能解が得られていない場合，AbsoluteGap / RelativeGap は +inf の値になっています．
- 実行可能解が得られていない場合，Objective は最大化問題の場合 -inf，最小化問題の場合 +inf の値になっています．
- 最適化計算情報の内，"MemoryAvailable" は Windows 版のみ有効です．
- コールバック関数内で例外が発生した場合は，その例外の stacktrace を標準エラー出力に表示し，分枝限定法を停止します．

サンプル

```
>>> def func(solver: dict[str, int | float]) -> bool:
...     if solver["SolutionCount"] >= 1:
...             # 実行可能解の個数が 1 以上になったら分枝限定法を停止する
...             return True
...     return False
>>> p = Problem()
>>> p.setCallback(mip_terminate=func)
```

solve(*type: Literal[min, max] = min*, *silent: bool = False*, *subprocess: bool = False*, *solfile: bool | str | None = None*, *iis: bool = True*) → *NuoptStatus*

求解を行います．引数の詳細は Problem の初期化時 help(Problem.__init__) で確認してください．引数は初期化時に設定した値に優先されます．省略した場合，初期化時の値が使用されます．

例外

- **RuntimeError** -- kernel of Nuorium Optimizer is terminated abnormally
- ***NuoptError*** -- internal error in Nuorium Optimizer
- ***SimpleError*** -- neither valid objective nor constraint in this model

戻り値

求解後のステータスです．

戻り値	ステータス
NuoptStatus.OPTIMAL	最適解が求まりました
NuoptStatus.FEASIBLE	実行可能解が求まりました
NuoptStatus.INFEASIBLE	実行不可能であることが判明しました
NuoptStatus.UNBOUNDED	非有界であることが判明しました
NuoptStatus.DUAL_INFEASIBLE	双対実行不可能（実行不可能あるいは非有界）であることが判明しました
NuoptStatus.ERROR	最適化計算実行時にエラーとなりました
NuoptStatus.UNKNOWN	上記以外のステータスとなりました

戻り値の型 *NuoptStatus*

注釈: NuoptStatus.OPTIMAL の以外の場合は，Problem.result.errorCode から詳細なエラー番号を取得することができます．また，Problem.result.errorMessage に最適化のカーネルからのエラーメッセージが設定されている場合があります．

参考:

pysimple.Problem, *pysimple.Problem.status*, *pysimple.Problem.result*, *pysimple.problem.Result.errorCode*, *pysimple.problem.Result.errorMessage*, *pysimple.NuoptStatus*

property status: ***pysimple.problem.NuoptStatus***

直前の Problem.solve() の結果を表す属性です．

- NuoptStatus.INITIAL : 求解をしたことがないことを表しています．
- NuoptStatus.OPTIMAL : 最適解が得られたことを表しています．
- NuoptStatus.FEASIBLE : 実行可能解が得られたことを表しています．
- NuoptStatus.INFEASIBLE : 実行可能解が存在しなかったと判定されたことを表しています．

- NuoptStatus.UNBOUNDED : 非有界であることを表しています.
- NuoptStatus.DUAL_INFEASIBLE : 双対実行不可能（実行不可能あるいは非有界）であることを表しています.
- NuoptStatus.UNKNOWN : 上記以外のステータスであることを表しています.

参考:

pysimple.NuoptStatus, *pysimple.Problem.solve*

property variables: pysimple.problem.IndexedDict[*pysimple.expression.Variable*]

問題に登録した制約式，目的関数に含まれる変数の一覧を表す属性です．変数名をキー，変数を値とする辞書です．参照には追加した順の番号も使用できます．負数の場合，逆順から参照されます．辞書は制約式，目的関数の登録状況，name 属性の変更等に対して動的に追従します.

例外 ***SimpleError*** -- Variable name {} is duplicated

サンプル

```
>>> x = Variable(name='x')
>>> y = Variable(name='y')
>>> p = Problem()
>>> p += x >= 1, 'cons1'
>>> p.variables
x:
x
>>> p += y >= 1, 'cons2'
>>> p.variables
x:
x
y:
y
>>> p.variables['y']
y:
y
>>> p.variables[1]
y:
y
>>> p.variables[-1]
y:
y
>>> del p['cons2']
>>> p.variables
x:
x
```

参考:

pysimple.Problem.objective, *pysimple.Problem.constraints*

バージョン 1.1.0 で追加: *mpsout()* が追加されました.

バージョン 1.3.0 で追加: *variables()* が追加されました.

バージョン 1.3.0 で追加: *__getitem__()*, *__delitem__()* に番号でアクセスできるようになりました.

バージョン 1.4.0 で追加: *setCallback()* が追加されました.

バージョン 1.6.0 で追加: *mpsout()* に引数 anonymous と initial が追加されました.

class pysimple.options.**ProblemOptions**

求解オプション定数を表すクラスです.

参考:

pysimple.Problem.options, *pysimple.Options*

__dir__() → list[str]

求解オプション定数一覧を表示します.

property branchCut: *pysimple.options.Cut* | None

導入される切除平面の数の目安.（分枝限定法専用）値: Cut.OFF, Cut.ON, Cut.AGGRESSIVE

property branchCutoff: float | None

足切り点.（分枝限定法専用）値: float

property branchDisconnected: *pysimple.options.Disconnected* | None

非連結成分検出.（分枝限定法専用）値: Disconnected.AUTO, Disconnected.OFF, Disconnected.ON

property branchDiving: *pysimple.options.Diving* | None

ヒューリスティクスサーチ Diving の頻度.（分枝限定法専用）値: Diving.AUTO, Diving.OFF, Diving.ON, Diving.AGGRESSIVE, Diving.SUPERAGGRESSIVE

property branchFeasPump: *pysimple.options.FeasPump* | None

ヒューリスティクスサーチ Feasibility Pump の頻度.（分枝限定法専用）値: FeasPump.AUTO, FeasPump.OFF, FeasPump.ON

property branchGapTolerance: float | None

上下界値のギャップの閾値. 絶対値で設定.（分枝限定法専用）値: float (-1 以上)

property branchMaxNode: int | None

探索問題数上限.（分枝限定法専用）値: int (-1 以上)

property branchMaxSolutionCount: int | None

整数解取得個数上限.（分枝限定法専用）値: int (-1 以上)

property branchNodeSelect: *pysimple.options.NodeSelect* | None

ノード選択.（分枝限定法専用）値: NodeSelect.AUTO, NodeSelect.BESTDEPTH, NodeSelect.BESTESTIMATE

property branchParallelMethod: *pysimple.options.ParallelMethod* | None

並列分枝限定法の手法．（分枝限定法専用）値: ParallelMethod.RACING, ParallelMethod.DETERMINISTIC_RACING, ParallelMethod.SUBTREE

property branchPresolve: *pysimple.options.Presolve* | None

分枝限定法における前処理．（分枝限定法専用）値: Presolve.AUTO, Presolve.OFF, Presolve.ON

property branchRelativeGapTolerance: float | None

上下界値のギャップの閾値．相対値で設定．（分枝限定法専用）値: float (-1 以上)

property branchRens: *pysimple.options.Rens* | None

ヒューリスティクスサーチ rens の導入．（分枝限定法専用）値: Rens.AUTO, Rens.OFF, Rens.ON

property branchRepairIteration: int | None

解の修復時の反復回収上限．（分枝限定法専用）値: int (0 以上)

property branchRepairSolution: *pysimple.options.RepairSolution* | None

ユーザ指定の初期解を元に解の修復をおこなう．（分枝限定法専用）値: RepairSolution.OFF, RepairSolution.ON, RepairSolution.AGGRESSIVE

property branchRins: *pysimple.options.Rins* | None

ヒューリスティクスサーチ rins の導入．（分枝限定法専用）値: Rins.AUTO, Rins.OFF, Rins.ON

property branchUseWcsp: *pysimple.options.UseWcsp* | None

ヒューリスティクスサーチ wcsp タブーサーチの導入．（分枝限定法専用）値: UseWcsp.AUTO, UseWcsp.OFF, UseWcsp.ON

property branchUseWls: *pysimple.options.UseWls* | None

ヒューリスティクスサーチ wls の導入．（分枝限定法専用）値: UseWls.AUTO, UseWls.OFF, UseWls.ON

property branchVariableSelectScore: *pysimple.options.VariableSelectScore* | None

分枝変数のスコアの評価方法．（分枝限定法専用）値: VariableSelectScore.AUTO, VariableSelectScore.SUM, VariableSelectScore.PRODUCT

property higherCrossover: bool | None

単体法へのクロスオーバー．（高次内点法専用）値: False, True

property hsimplexMethod: *pysimple.options.Options.HsimplexMethod* | None

アルゴリズム hsimplex の具体的な手法．（hsimplex 専用）値: HsimplexMethod.AUTO, HsimplexMethod.PRIMAL, HsimplexMethod.DUAL

property kktEps: float | None

KKT 条件の残差停止条件．値: float (0 より大きい かつ 0.0001 以下)

property maxIteration: int | None

最大反復回数．（内点法，wcsp，wls）値: int

property maxMemory: int | None

メモリ利用量上限（MiB）．負の値の場合は，分枝限定法は残り利用可能メモリ（MiB）による制限，wls は無制限と解釈します．（分枝限定法，wls）値: int (-1 以上)

property maxTime: float | None

計算時間上限（秒）.（分枝限定法，wcsp，wls）値: float (-1 以上)

property method: *pysimple.options.Options.Method* | None

求解アルゴリズムの種類．値: Method.AUTO, Method.LIPM, Method.HIGHER, Method.TIPM, Method.BFGS, Method.SIMPLEX, Method.ASQP, Method.LSQP, Method.TSQP, Method.HSIMPLEX, Method.WCSP, Method.WLS

property objectiveTarget: float | None

分枝限定法および wls では目的関数の目標値．目標値よりも良い解を発見した場合は計算を終了します．wcsp ではこの値を目的関数の下 (上) 限とするソフト制約と解釈します．（分枝限定法，wcsp，wls）値: float

property randomSeed: int | None

乱数のシード．（wcsp 専用）値: int (1 以上)

property scaling: *pysimple.options.Options.Scaling* | None

スケーリングの種類．値: Scaling.AUTO, Scaling.OFF, Scaling.ON, Scaling.MINMAX, Scaling.CR

property simplexDualTolerance: float | None

双対変数の実行可能性判定閾値．（単体法専用）値: float (0 以上 かつ 0.0001 以下)

property simplexPrimalTolerance: float | None

主変数の実行可能性判定閾値．（単体法専用）値: float (0 以上 かつ 0.0001 以下)

property threads: int | None

スレッド数．-1 の場合は自動的に設定します．0 は 1 と解釈します．（分枝限定法，wcsp，wls）値: int (-1 以上)

property tryCount: int | None

試行回数．（wcsp，wls）値: int (1 以上)

property wcspInitialValueActivation: bool | None

初期値からの探索．（wcsp 専用）値: False, True

property wcspPhaseOneMaxIteration: int | None

制約充足フェーズにおける制約充足フェーズにおける反復回数上限．（wcsp 専用）値: int (-1 以上)

property wcspPhaseOneMaxTime: int | None

制約充足フェーズにおける計算時間上限．（wcsp 専用）値: int (-1 以上)

property wcspPhaseTwoMaxIntervalIteration: int | None

解更新フェーズにおける解更新間隔反復回数上限．（wcsp 専用）値: int (-1 以上)

property wcspPhaseTwoMaxIntervalTime: int | None

解更新フェーズにおける計算時間上限．（wcsp 専用）値: int (-1 以上)

class pysimple.**Options**

ソルバのパラメータを設定するためのクラスです．デフォルト値は Nuorium Optimizer SIMPLE マニュアルの「5. 求解オプション設定」を参照してください．

参考:

pysimple.Problem.options, *pysimple.options.ProblemOptions*

class Branch

分枝限定法の求解オプション定数を表すクラスです.

class Cut(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

導入される切除平面の数の目安.（分枝限定法専用）

AGGRESSIVE = 2

OFF = 0

ON = 1

class Disconnected(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

非連結成分検出.（分枝限定法専用）

AUTO = -1

OFF = 0

ON = 1

class Diving(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

ヒューリスティクスサーチ Diving の頻度.（分枝限定法専用）

AGGRESSIVE = 2

AUTO = -1

OFF = 0

ON = 1

SUPERAGGRESSIVE = 3

class FeasPump(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

ヒューリスティクスサーチ Feasibility Pump の頻度.（分枝限定法専用）

AUTO = -1

OFF = 0

ON = 1

class NodeSelect(*value*, *names=None*, *, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

ノード選択.（分枝限定法専用）

AUTO = -1

BESTDEPTH = 1

BESTESTIMATE = 2

class ParallelMethod(*value*, *names=None*, *, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

並列分枝限定法の手法.（分枝限定法専用）

DETERMINISTIC_RACING = 1

RACING = 0

SUBTREE = 2

class Presolve(*value*, *names=None*, *, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

分枝限定法における前処理.（分枝限定法専用）

AUTO = -1

OFF = 0

ON = 1

class Rens(*value*, *names=None*, *, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

ヒューリスティクスサーチ rens の導入.（分枝限定法専用）

AUTO = -1

OFF = 0

ON = 1

class RepairSolution(*value*, *names=None*, *, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

ユーザ指定の初期解を元に解の修復をおこなう.（分枝限定法専用）

AGGRESSIVE = 2

OFF = 0

ON = 1

class Rins(*value*, *names=None*, *, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

ヒューリスティクスサーチ rins の導入.（分枝限定法専用）

AUTO = -1

OFF = 0

ON = 1

class UseWcsp(*value*, *names=None*, *, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

ヒューリスティクスサーチ wcsp タブーサーチの導入.（分枝限定法専用）

AUTO = -1

OFF = 0

ON = 1

class UseWls(*value*, *names=None*, *, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

ヒューリスティクスサーチ wls の導入.（分枝限定法専用）

AUTO = -1

OFF = 0

ON = 1

class VariableSelectScore(*value*, *names=None*, *, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

分枝変数のスコアの評価方法.（分枝限定法専用）

AUTO = -1

PRODUCT = 1

SUM = 0

class HsimplexMethod(*value*, *names=None*, *, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

アルゴリズム hsimplex の具体的な手法を表すクラスです.

AUTO = -1

DUAL = 1

PRIMAL = 0

class Method(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

求解アルゴリズムの求解オプション定数を表すクラスです.

ASQP = 'asqp'

AUTO = 'auto'

BFGS = 'bfgs'

HIGHER = 'higher'

HSIMPLEX = 'hsimplex'

LIPM = 'lipm'

LSQP = 'lsqp'

SIMPLEX = 'simplex'

TIPM = 'tipm'

TSQP = 'tsqp'

WCSP = 'wcsp'

WLS = 'wls'

class Scaling(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

スケーリングの求解オプション定数を表すクラスです.

AUTO = -1

CR = 3

MINMAX = 2

OFF = 0

ON = 1

バージョン 1.4.0 で追加: 求解オプション定数のクラス構造が変更されました.

class pysimple.**NuoptStatus**

ベースクラス: enum.IntEnum

ソルバの計算後の状態値を定義しているクラスです.

参考:

pysimple.Problem.solve

DUAL_INFEASIBLE = 5

ERROR = 7

FEASIBLE = 2

INFEASIBLE = 3

INITIAL = 0

OPTIMAL = 1

UNBOUNDED = 4

UNKNOWN = 6

class pysimple.problem.**Result**

求解後の求解情報を表すクラスです.

property consInfeasibility: float

制約式の実行不可能性.

property elapseTime: float

計算時間．Nuorium Optimizer のカーネルが最適化計算を行っている時間です．単位は秒です.

property errorCode: int

Nuorium Optimizer のエラーコード．最適解が得られた場合は 0 が設定されています．エラーコードは Nuorium Optimizer SIMPLE マニュアルの「A.1.1 Nuorium Optimizer のエラー/警告メッセージ」を参照してください.

property errorMessage: str

エラーメッセージ．Nuorium Optimizer がエラーとなった場合に，それに対応するエラーメッセージが設定されています.

property factCount: int

行列分解の回数．アルゴリズムとして内点法を選択した時のみ意味を持ちます.

property fevals: int

制約式の評価回数．アルゴリズムとして内点法を選択した時のみ意味を持ちます.

property hardPenalty: float

ハード制約のペナルティ (重み×違反量) の合計値.

property iis: *pysimple.problem.IIS*

問題が実行不可能な場合の IIS 情報．IIS 情報がない場合は空の IIS オブジェクトが設定されています.

property infeasibility: float

実行不可能性．スケーリングを行っている場合はスケールを戻して計算した値になります.

property iters: int

反復回数．アルゴリズムとして内点法を選択した時のみ意味を持ちます.

property method: str

求解に用いられた解法.

property nfunc: int

制約式と目的関数の数．目的関数は必ず 1 であるため，「制約式の数+1」が設定されています.

property nvars: int

変数の数.

property optValue: float

目的関数の値．最適解の場合は最適値となります.

property peakPhysicalMemoryUsed: int

最適化計算で利用された最大物理メモリ量です．単位は MiB です.

property peakVirtualMemoryUsed: int

最適化計算で利用された最大仮想メモリ量です．単位は MiB です.

property residual: float

KKT 条件の充足度合い．連続変数の最適化時のみ意味を持ちます.

property semiHardPenalty: float

セミハード制約のペナルティ (重み×違反量) の合計値.

property softPenalty: float

ソフト制約のペナルティ (重み×違反量) の合計値.

property tolerance: float

反復停止条件として実際にどの値が使われたか．アルゴリズムとして内点法を選択した時のみ意味を持ちます.

property varInfeasibility: float

変数の実行不可能性.

class pysimple.problem.**IIS**

ベースクラス: dict[int, IIS.OneIIS]

IIS 情報を保持するクラスです．制約式は 0 始まりのインデックスで管理されており，そのインデックスを使い各制約式情報にアクセスできます.

サンプル

```
>>> x = Variable(name='x', lb=0)
>>> y = Variable(name='y', lb=0, ub=1)
>>> z = Variable(name='z', lb=0)
>>> p = Problem()
>>> p += x + y + z
>>> p += y + z >= 0, 'cons1'
```

(次のページに続く)

(前のページからの続き)

```
>>> p += x + y >= 5, 'cons2'
>>> p += x + z <= 3, 'cons3'
>>> p.solve(silent=True)
<NuoptStatus.INFEASIBLE: 3>
>>> p.result.iis
  0: cons2: violation=-1
      x+y>=5
  1: cons3
      -x-z>=-3
>>> p.result.iis[0]
  0: cons2: violation=1
      x+y>=5
          x: value=3: dual=0: [0, inf]
          y: value=1: dual=-1: [0, 1]
>>> del p[p.result.iis[0].consname]  # index=0 の制約式を delete
>>> p
Problem(name='Problem', type=min, silent=False, subprocess=False, solfile=None,␣
↪iis=True):
[constraints]
cons1:
y+z>=0
cons3:
-x-z>=-3
.
[objective]
((x+y)+z):
x+y+z
>>> p.solve(silent=True)
<NuoptStatus.OPTIMAL: 1>
>>> print(p.result.iis)
no IIS
```

参考:

pysimple.problem.IIS.OneIIS

class OneIIS

IIS 情報のうち 1 つの制約式の情報を保持するクラスです.

property OneIIS.**consname**: str

制約式名.

property OneIIS.**dual**: int | float

制約式の dual 値.

property OneIIS.**formatted**: str

制約式の展開された式表示.

property OneIIS.**key**: int | float | str | tuple[int | float | str, ...]

制約式の添字.

property OneIIS.**no**: int

制約式番号 (0 始まり).

property OneIIS.**variables**: list[*pysimple.expression.Variable*]

制約式に含まれる変数列.

property OneIIS.**violation**: int | float

制約式の違反量.

5.1.7 シリアライズ

class pysimple.serialize.**Serialize**

PySIMPLE オブジェクトをシリアライズするためのクラスです. pickle.dump/dumps でシリアライズ可能なオブジェクトに加え, PySIMPLE オブジェクトをシリアライズできます.

static dump(*obj: Any*, *file: BinaryIO*) → None

オブジェクトをバイナリにシリアライズしてファイル出力します.

パラメータ

- **obj** (*Any*) -- シリアライズするオブジェクト.
- **file** (*BinaryIO*) -- 書き込むファイルオブジェクト

サンプル

```
>>> x = Variable(name='x')
>>> with open('dump.pkl', 'wb') as f:
...     Serialize.dump(x, f)
...
>>> with open('dump.pkl', 'rb') as g:
...     x_ = Serialize.load(g)
...
```

参考:

Serialize.dumps, *Serialize.load*

static dumps(*obj: Any*) → Pickler

オブジェクトをバイナリにシリアライズして返します.

パラメータ **obj** (*Any*) -- シリアライズするオブジェクト.

戻り値 pickle.Pickler をラップしたバイナリを保持するオブジェクトです.

戻り値の型 *Pickler*

サンプル

```
>>> x = Variable(name='x')
>>> pkl = Serialize.dumps(x)
>>> pkl
Pickler(version_info(major=1, minor=3, micro=0))
>>> x_ = Serialize.loads(pkl)
```

参考:

Serialize.dump, *Serialize.loads*

static load(*file: BinaryIO*) → Any

シリアライズされたオブジェクトをバイナリファイルから復元します.

パラメータ **file** (*BinaryIO*) -- シリアライズしたバイナリを出力するファイルオブジェクト.

戻り値 シリアライズを復元したオブジェクト.

戻り値の型 *Any*

サンプル

```
>>> x = Variable(name='x')
>>> with open('dump.pkl', 'wb') as f:
...     Serialize.dump(x, f)
...
>>> with open('dump.pkl', 'rb') as g:
...     x_ = Serialize.load(g)
...
```

参考:

Serialize.dump, *Serialize.loads*

static loads(*data: Pickler*) → Any

シリアライズされたオブジェクトを pysimple.Pickler オブジェクトから復元します.

パラメータ **data** (*Pickler*) -- シリアライズした Pickler オブジェクト.

戻り値 シリアライズを復元したオブジェクト.

戻り値の型 *Any*

サンプル

```
>>> x = Variable(name='x')
>>> pkl = Serialize.dumps(x)
>>> pkl
Pickler(version_info(major=1, minor=3, micro=0))
>>> x_ = Serialize.loads(pkl)
```

参考:

Serialize.dumps, *Serialize.load*

5.2 関数

出力関数を除く関数では引数にキーワード引数を用いることはできません.

5.2.1 条件関数

pysimple.**Condition**(*elm: ELEMENT | tuple[ELEMENT, ...]*, *cond:* Cond | *tuple[*Cond, ...*]*) → *Cond*

条件式を表す Cond 型を生成する関数です.

パラメータ

- **elm** (*ELEMENT* or tuple of them) -- Element の列を与えます.
- **cond** (Cond *or tuple of them*) -- 条件式の列を与えます.

戻り値の型 *Cond*

注釈: i!=2 や (1<i) & (i<3) のような簡単な場合は Condition 関数を用いなくても構いません.

サンプル

```
>>> i = Element(value=[1,2,3], name='i')
>>> Condition((i, i), i!=2)
((i,i), (i!=2))[i,i] in [(1, 1), (3, 3)]
>>> Condition(i, (1<i, i<3))
(i, ((i>1),(i<3)))[i] in [2]
>>> ij = Element(value=[(1,3), (1,4), (2,3)], name='ij')
>>> Condition(ij, ij(1)!=4)
(ij, (ij(1)!=4))[ij] in [(1, 3), (2, 3)]
```

参考:

pysimple.condition.Cond, *pysimple.condition.Cond.__and__*

5.2.2 出力関数

pysimple.**Printf**(*format_string: str*, **args: Any*, ***kwds: Any*) → None

書式指定文字列を標準出力に出力する関数です．Python の書式指定文字列の文法に加えて，PySIMPLE のオブジェクトに対応しています．

パラメータ

- **format_string** (*str*) -- 書式を指定する文字列です．文法は「書式指定文字列の文法」を参照してください．
- ***args** (*Any*) -- キーワードなし引数です．
- ****kwds** (*Any*) -- キーワードあり引数です．

サンプル

```
>>> i = Element(value=[1,2], name='i')
>>> a = Parameter(index=i, value={1: 10, 2: 20}, name='a')
>>> Printf('{}[{}] = {}', a.name, i, a[i])
a[1] = 10
a[2] = 20
>>> Printf('{name}[{}] = {value}', i, name=a.name, value=a[i])
a[1] = 10
a[2] = 20
>>> x = Variable(index=i, name='x')
>>> Printf('{}: ({}=){} >= {}', x[i]>=a[i], x[i], x[i].val, a[i])
x[1]>=10: (x[1]=)0 >= 10
x[2]>=20: (x[2]=)0 >= 20
```

参考:

pysimple.Fprintf

pysimple.**Fprintf**(*file_like: TextIO*, *format_string: str*, **args: Any*, ***kwds: Any*) → None

書式指定文字列をファイルに出力する関数です．Python の書式指定文字列の文法に加えて，PySIMPLE のオブジェクトに対応しています．

パラメータ

- **file_like** (*TextIO*) -- ファイルオブジェクト，sys.stdout などの出力先を指定するオブジェクトです．
- **format_string** (*str*) -- 書式を指定する文字列です．文法は「書式指定文字列の文法」を参照してください．

- ***args** (*Any*) -- キーワードなし引数です.
- ****kwds** (*Any*) -- キーワードあり引数です.

サンプル

```
>>> import sys
>>> i = Element(value=[1,2], name='i')
>>> a = Parameter(index=i, value={1: 10, 2: 20}, name='a')
>>> Fprintf(sys.stdout, '{}[{}] = {}', a.name, i, a[i])
a[1] = 10
a[2] = 20
>>> Fprintf(sys.stdout, '{name}[{}] = {avalue}', i, name=a.name, avalue=a[i])
a[1] = 10
a[2] = 20
```

参考:

pysimple.Printf

5.2.3 数学関数

数学関数．範囲関数，一変数関数，二変数関数，多変数関数に分かれる．

pysimple.func.**Acos**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.acos (obj) の結果を，それ以外の場合は math.acos を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Acosh**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.acosh (obj) の結果を，それ以外の場合は math.acosh を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Asin**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.asin (obj) の結果を，それ以外の場合は math.asin を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Asinh**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.asinh (obj) の結果を，それ以外の場合は math.asinh を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Atan**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.atan (obj) の結果を，それ以外の場合は math.atan を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Atan2**(*obj1: ObjPrm*, *obj2: ObjPrm*) → Num | *Parameter*

パラメータ

- **obj1** (*ObjPrm*) --
- **obj2** (*ObjPrm*) --

戻り値 obj1, obj2 が *DType* の場合は math.atan2 (obj1, obj2) の結果を，それ以外の場合は math.atan2 を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Atanh**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.atanh (obj) の結果を，それ以外の場合は math.atanh を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Ceil**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.ceil (obj) の結果を，それ以外の場合は math.ceil を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Cos**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.cos (obj) の結果を，それ以外の場合は math.cos を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Cosh**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.cosh (obj) の結果を，それ以外の場合は math.cosh を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Erf**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.erf (obj) の結果を，それ以外の場合は math.erf を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Exp**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.exp (obj) の結果を，それ以外の場合は math.exp を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Fabs**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.fabs (obj) の結果を，それ以外の場合は math.fabs を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Floor**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.floor (obj) の結果を，それ以外の場合は math.floor を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Fmod**(*obj1: ObjPrm*, *obj2: ObjPrm*) → Num | *Parameter*

pysimple.func.**Hypot**(*obj1: ObjPrm*, *obj2: ObjPrm*) → Num | *Parameter*

パラメータ

- **obj1** (*ObjPrm*) --
- **obj2** (*ObjPrm*) --

戻り値 obj1, obj2 が *DType* の場合は math.hypot (obj1, obj2) の結果を，それ以外の場合は math.hypot を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Log**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.log (obj) の結果を，それ以外の場合は math.log を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Log10**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.log10 (obj) の結果を，それ以外の場合は math.log10 を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Max**(*obj: ObjPrm*, *idx: ELEMENT | tuple[ELEMENT, ...] | Index | None = None*) → DType | *Parameter*

obj の範囲 idx 上での最大値を計算したオブジェクトを返します.

パラメータ

- **obj** (*ObjPrm*) -- 最大値を計算する対象のオブジェクトです.
- **idx** (None or *ELEMENT* or tuple of them or Index) -- 最大値を計算する範囲です．省略された場合，obj のすべての範囲についての最大値を計算します．同じ Element を複数回含めることはできません.

戻り値 添字が残る場合は Parameter を，残らない場合は *DType* を返します.

戻り値の型 *DType* or Parameter

例外

- **ValueError** -- duplicate Element in range of {}
- **ValueError** -- Max arg is an empty index

注釈: obj や idx の添字範囲が空集合の場合は ValueError を投げます．idx が空タプルの場合は obj と等価なオブジェクトを返します.

サンプル

```
>>> I = Set(value=[1,2]); i = Element(set=I, name='i')
>>> J = Set(value='XY'); j = Element(set=J, name='j')
>>> a = Parameter(index=(i,j), value={(ii, jj): ii*10+J.index(jj) for ii, jj in␣
↪I*J}, name='a')
>>> a
a[1,'X']=10
a[1,'Y']=11
a[2,'X']=20
a[2,'Y']=21
>>> Max(a[i,j], i)
Max(a[i,j], i)['X']=20
Max(a[i,j], i)['Y']=21
>>> Max(a[i,j], j)
Max(a[i,j], j)[1]=11
Max(a[i,j], j)[2]=21
>>> Max(a[i,j], (i,j))
21
>>> Max(a[i,j])
21
```

参考:

Min, *MaxOf*

pysimple.func.**MaxOf**(**args: ObjPrm*) → DType | *Parameter*

オブジェクトの各最大値を返します．例えば変数の下限を一度に設定するときに便利です．

パラメータ ***args** (*ObjPrm*) -- オブジェクトの列．1 つ以上必要です．

例外 **ValueError** -- max() arg is an empty sequence

サンプル

```
>>> i = Element(value=[1,2,3], name='i')
>>> a = Parameter(index=i, value={1: 2, 2: 3, 3: 1}, name='a')
>>> b = Parameter(index=i, value={1: 3, 2: 1, 3: 2}, name='b')
>>> Printf('i={i}, a={a}, b={b}', i=i, a=a[i], b=b[i])
i=1, a=2, b=3
i=2, a=3, b=1
i=3, a=1, b=2
>>> MaxOf(i, a[i], b[i])
MaxOf(i, a[i], b[i])[1]=3
MaxOf(i, a[i], b[i])[2]=3
MaxOf(i, a[i], b[i])[3]=3
```

(次のページに続く)

(前のページからの続き)

```
>>> x = Variable(index=i, name='x')
>>> x[i] >= MaxOf(i, a[i], b[i])
(x[i]>=MaxOf(i, a[i], b[i])[i]):
x[1]>=3
x[2]>=3
x[3]>=3
```

参考:

Max, *MinOf*

pysimple.func.**Min**(*obj: ObjPrm*, *idx: ELEMENT | tuple[ELEMENT, ...] | Index | None = None*) → DType | *Parameter*

obj の範囲 idx 上での最小値を計算したオブジェクトを返します.

パラメータ

- **obj** (*ObjPrm*) -- 最小値を計算する対象のオブジェクトです.
- **idx** (None or *ELEMENT* or tuple of them or Index) -- 最小値を計算する範囲です. 省略された場合, obj のすべての範囲についての最小値を計算します. 同じ Element を複数回含めることはできません.

戻り値 添字が残る場合は Parameter を, 残らない場合は *DType* を返します.

戻り値の型 *DType* or Parameter

例外

- **ValueError** -- duplicate Element in range of {}
- **ValueError** -- Min arg is an empty index

注釈: obj や idx の添字範囲が空集合の場合は ValueError を投げます. idx が空タプルの場合は obj と等価なオブジェクトを返します.

サンプル

```
>>> I = Set(value=[1,2]); i = Element(set=I, name='i')
>>> J = Set(value='XY'); j = Element(set=J, name='j')
>>> a = Parameter(index=(i,j), value={(ii, jj): ii*10+J.index(jj) for ii, jj in␣
↪I*J}, name='a')
>>> a
a[1,'X']=10
a[1,'Y']=11
a[2,'X']=20
a[2,'Y']=21
```

(次のページに続く)

(前のページからの続き)

```
>>> Min(a[i,j], i)
Min(a[i,j], i)['X']=10
Min(a[i,j], i)['Y']=11
>>> Min(a[i,j], j)
Min(a[i,j], j)[1]=10
Min(a[i,j], j)[2]=20
>>> Min(a[i,j], (i,j))
10
>>> Min(a[i,j])
10
```

参考:

Max, *MinOf*

pysimple.func.**MinOf**(**args: ObjPrm*) → DType | *Parameter*

オブジェクトの各最小値を返します．例えば変数の上限を一度に設定するときに便利です．

パラメータ ***args** (*ObjPrm*) -- オブジェクトの列．1 つ以上必要です．

例外 **ValueError** -- min() arg is an empty sequence

サンプル

```
>>> i = Element(value=[1,2,3], name='i')
>>> a = Parameter(index=i, value={1: 2, 2: 3, 3: 1}, name='a')
>>> b = Parameter(index=i, value={1: 3, 2: 1, 3: 2}, name='b')
>>> Printf('i={i}, a={a}, b={b}', i=i, a=a[i], b=b[i])
i=1, a=2, b=3
i=2, a=3, b=1
i=3, a=1, b=2
>>> MinOf(i, a[i], b[i])
MinOf(i, a[i], b[i])[1]=1
MinOf(i, a[i], b[i])[2]=1
MinOf(i, a[i], b[i])[3]=1
>>> x = Variable(index=i, name='x')
>>> x[i] <= MinOf(i, a[i], b[i])
(x[i]<=MinOf(i, a[i], b[i])[i]):
-x[1]>=-1
-x[2]>=-1
-x[3]>=-1
```

参考:

Min, *MaxOf*

pysimple.func.**Prod**(*obj: ObjPrm*, *idx: ELEMENT* | *tuple[ELEMENT, ...]* | Index | *None* = *None*) → DType | *Parameter*

obj の範囲 idx 上での積を計算したオブジェクトを返します.

パラメータ

- **obj** (*ObjPrm*) -- 積を計算する対象のオブジェクトです.
- **idx** (None or *ELEMENT* or tuple of them or Index) -- 積を計算する範囲です. 省略された場合, obj のすべての範囲についての積を計算します. 同じ Element を複数回含めることはできません.

戻り値 添字が残る場合は Parameter を, 残らない場合は *DType* を返します.

戻り値の型 *DType* or Parameter

例外 **ValueError** -- duplicate Element in range of {}

注釈: obj や idx の添字範囲が空集合の場合は 1 を返します. idx が空タプルの場合は obj と等価なオブジェクトを返します.

サンプル

```
>>> I = Set(value=[1,2]); i = Element(set=I, name='i')
>>> J = Set(value='XY'); j = Element(set=J, name='j')
>>> a = Parameter(index=(i,j), value={(ii, jj): ii*10+J.index(jj) for ii, jj in␣
↪I*J}, name='a')
>>> a
a[1,'X']=10
a[1,'Y']=11
a[2,'X']=20
a[2,'Y']=21
>>> Prod(a[i,j], i)
Prod(a[i,j], i)['X']=200
Prod(a[i,j], i)['Y']=231
>>> Prod(a[i,j], j)
Prod(a[i,j], j)[1]=110
Prod(a[i,j], j)[2]=420
>>> Prod(a[i,j], (i,j))
46200
>>> Prod(a[i,j])
46200
```

参考:

Sum, *Min*, *Max*

pysimple.func.**Sin**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.sin (obj) の結果を，それ以外の場合は math.sin を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Sinh**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.sinh (obj) の結果を，それ以外の場合は math.sinh を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Sqrt**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.sqrt (obj) の結果を，それ以外の場合は math.sqrt を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Sum**(*obj: ObjPrm*, *idx:* Index*[ELEMENT]* | *None* = *None*, */*) → DType | *Parameter*

pysimple.func.**Sum**(*obj: VarExp*, *idx:* Index*[ELEMENT]* | *None* = *None*, */*) → *Expression*

Sum(obj: ObjExp, idx: ELEMENT | tuple[ELEMENT, ...] | Index | None=None) -> DType | Parameter | Expression

obj の範囲 idx 上での和を計算したオブジェクトを返します.

パラメータ

- **obj** (*ObjExp*) -- 和を計算する対象のオブジェクトです.
- **idx** (None or *ELEMENT* or tuple of them or Index) -- 和を計算する範囲です．省略された場合，obj のすべての範囲についての和を計算します．同じ Element を複数回含めることはできません.

戻り値 obj が Variable, Expression の場合は Expression を，それ以外で添字が残る場合は Parameter を，残らない場合は *DType* を返します.

戻り値の型 *DType* or Parameter or Expression

例外 **ValueError** -- duplicate Element in range of {}

注釈: obj や idx の添字範囲が空集合の場合は 0 を返します．idx が空タプルの場合は obj と等価なオブジェクトを返します.

サンプル

```
>>> I = Set(value=[1,2]); i = Element(set=I, name='i')
>>> J = Set(value='XY'); j = Element(set=J, name='j')
>>> a = Parameter(index=(i,j), value={(ii, jj): ii*10+J.index(jj) for ii, jj in␣
↪I*J}, name='a')
>>> a
a[1,'X']=10
a[1,'Y']=11
a[2,'X']=20
a[2,'Y']=21
>>> Sum(a[i,j], i)
Sum(a[i,j], i)['X']=30
Sum(a[i,j], i)['Y']=32
>>> Sum(a[i,j], j)
Sum(a[i,j], j)[1]=21
Sum(a[i,j], j)[2]=41
>>> Sum(a[i,j], (i,j))
62
>>> Sum(a[i,j])
62
>>> x = Variable(index=(i,j), name='x')
>>> Sum(x[i,j], i)
Sum(x[i,j], i):
x[1,'X']+x[2,'X']
x[1,'Y']+x[2,'Y']
>>> Sum(x[i,j], j)
Sum(x[i,j], j):
x[1,'X']+x[1,'Y']
x[2,'X']+x[2,'Y']
>>> Sum(x[i,j], (i,j))
Sum(x[i,j], (i,j)):
x[1,'X']+x[1,'Y']+x[2,'X']+x[2,'Y']
```

参考:

Selection, *Prod*, *Min*, *Max*

pysimple.func.**Tan**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.tan (obj) の結果を，それ以外の場合は math.tan を各要素に適用した Parameter を返す.

戻り値の型 *Num* or Parameter

pysimple.func.**Tanh**(*obj: ObjPrm*) → Num | *Parameter*

パラメータ **obj** (*ObjPrm*) --

戻り値 obj が *DType* の場合は math.tanh (obj) の結果を，それ以外の場合は math.tanh を各要素に適用した Parameter を返す．

戻り値の型 *Num* or Parameter

5.2.4 選択関数

pysimple.func.**Selection**(*obj:* BinaryVariable, *idx: ELEMENT* | *tuple[ELEMENT, ...]* | Index | *None* = *None*) → *SelectionConstraint*

添字つき 0-1 整数変数の中で一つだけを 1 に固定したい場合に用います．wcsp を解法とする場合，内部的には複数の 0-1 整数変数を用意する替わりに一つの離散変数を用意するため，内部処理が高速化されます．それ以外では Sum を用いた書き換えと等価です．

パラメータ

- **obj** (BinaryVariable) -- 計算する対象の 0-1 整数変数です．添字づけられている必要があります．
- **idx** (None or *ELEMENT* or tuple of them or Index) -- 計算する範囲です．省略された場合，obj のすべての範囲を対象とします．同じ Element を複数回含めることはできません．

戻り値の型 *SelectionConstraint*

例外

- **ValueError** -- duplicate Element in range of {}
- **ValueError** -- the argument of Selection should be indexed variable
- **ValueError** -- Selection arg is an empty index

サンプル

```
>>> i = Element(value=[1, 2], name='i')
>>> j = Element(value=[3, 4], name='j')
>>> z = BinaryVariable(index=(i,j), name='z')
>>> Selection(z[i,j], i)
Selection(z[i,j], i):
z[1,3]+z[2,3]==1
z[1,4]+z[2,4]==1
>>> Selection(z[i,j], j)
Selection(z[i,j], j):
z[1,3]+z[1,4]==1
z[2,3]+z[2,4]==1
>>> Selection(z[i,j], (i,j))
Selection(z[i,j], (i,j)):
z[1,3]+z[1,4]+z[2,3]+z[2,4]==1
```

参考:

Sum

5.2.5 制約関数

pysimple.constraint.**HardConstraint**() → *ConstraintWeight*

制約式がハード制約であることを表します.

戻り値の型 *ConstraintWeight*

pysimple.constraint.**SemiHardConstraint**() → *ConstraintWeight*

制約式がセミハード制約であることを表します.

戻り値の型 *ConstraintWeight*

pysimple.constraint.**SoftConstraint**(*weight: int*, *quad: float | None = None*, *linear: float | None = None*) → *ConstraintWeight*

制約式がソフト制約であることを表します.

パラメータ

- **weight** (*non-negative int or -1 or -2*) -- 重みの基本となるパラメータです.
- **quad** (*non-negative float*) -- 重みの二次の係数です.
- **linear** (*non-negative float*) -- 重みの一次の係数です.

戻り値の型 *ConstraintWeight*

5.3 例外

exception pysimple.**SimpleError**(*msg: str*)

ベースクラス: Exception

数理最適化のモデリング特有の例外を表すクラスです.

参考:

pysimple.NuoptError

サンプル

```
>>> ij = Element(value=[(1,3), (1,4), (2,3)], name='ij')
>>> ij + 1
pysimple.error.SimpleError: dim of 'Element' must be 1
>>> i = Element(value=[1,2], name='i')
>>> x = Variable(index=i, name='x')
>>> x + 1
pysimple.error.SimpleError: Variable 'x' needs to be with index (no index given)
>>> Sum(x[i], i) + i
pysimple.error.SimpleError: illegal use of Element 'i' which has been operated␣
↪already
>>> Condition(ij, i>1)
pysimple.error.SimpleError: illegal use of Element 'i'
>>> p = Problem()
>>> p += x[i]
pysimple.error.SimpleError: objective cannot be defined with index
>>> p += Sum(x[i])
>>> p += Sum(x[i])
pysimple.error.SimpleError: objective can only be assigned once
>>> p += x[i] >= 1
>>> p += x[i] >= 1
pysimple.error.SimpleError: override constraint '(x[i]>=1)'
```

exception pysimple.**NuoptError**

ベースクラス: Exception

Nuorium Optimizer の内部で発生した例外を表すクラスです.

参考:

pysimple.SimpleError

5.4 演算

5.4.1 単項演算

pysimple.**__pos__**(*obj: ObjExp*) → *Parameter* | *Expression*

単項演算 +obj を返します.

パラメータ **obj** (*ObjExp*) --

戻り値 obj が *ObjPrm* の場合は Parameter を，それ以外の場合は Expression を返します.

戻り値の型 *Parameter* or *Expression*

pysimple.**__neg__**(*obj: ObjExp*) → *Parameter* | *Expression*

単項演算 -obj を返します.

パラメータ **obj** (*ObjExp*) --

戻り値 obj が *ObjPrm* の場合は Parameter を，それ以外の場合は Expression を返します.

戻り値の型 *Parameter* or *Expression*

5.4.2 算術演算

Set.**__sub__**(*rset: Iterable[DType | Key]*) → *Set*

Set に含まれて rset に含まれない集合を返します.

パラメータ **rset** (Iterable[*DType* | *Key*]) -- Set や set でなくても構いません.

戻り値の型 *Set*

サンプル

```
>>> I = Set(value=[1,2,3,4], name='I')
>>> J = Set(value=[2,4,6,8], name='J')
>>> I - J
Set(name='(I-J)', value=[1, 3])
>>> I - {2,4,6,8}
Set(name='(I-{8, 2, 4, 6})', value=[1, 3])
>>> I - [2,4,6,8]
Set(name='(I-[2, 4, 6, 8])', value=[1, 3])
```

Set.**__mul__**(*rset: Iterable[DType | Key]*) → *Set*

Set と rset の直積集合を返します.

パラメータ **rset** (Iterable[*DType* | *Key*]) -- Set や set でなくても構いません.

戻り値の型 *Set*

サンプル

```
>>> I = Set(value=[1,2], name='I')
>>> I * I
Set(name='(I*I)', dim=2, value=[(1, 1), (1, 2), (2, 1), (2, 2)])
>>> I * {3,4}
Set(name='(I*{3, 4})', dim=2, value=[(1, 3), (1, 4), (2, 3), (2, 4)])
>>> I * [3,4]
Set(name='(I*[3, 4])', dim=2, value=[(1, 3), (1, 4), (2, 3), (2, 4)])
```

(次のページに続く)

(前のページからの続き)

```
>>> I * I * I
Set(name='((I*I)*I)', dim=3, value=[(1, 1, 1), (1, 1, 2), (1, 2, 1), (1, 2, 2),␣
↪(2, 1, 1), (2, 1, 2), (2, 2, 1), (2, 2, 2)])
```

Set.**__rsub__**(*lset: Iterable*) → *Set*

lset に含まれて Set に含まれない集合を返します.

パラメータ **lset** (Iterable[*DType* | *Key*]) -- Set や set でなくても構いません.

戻り値の型 *Set*

サンプル

```
>>> I = Set(value=[1,2,3,4], name='I')
>>> {2,4,6,8} - I
Set(name='({8, 2, 4, 6}-I)', value=[8, 6])
>>> [2,4,6,8] - I
Set(name='([2, 4, 6, 8]-I)', value=[6, 8])
```

Set.**__rmul__**(*lset: Iterable[DType | Key]*) → *Set*

lset と Set の直積集合を返します.

パラメータ **lset** (Iterable[*DType* | *Key*]) -- Set や set でなくても構いません.

戻り値の型 *Set*

サンプル

```
>>> I = Set(value=[1,2], name='I')
>>> {3,4} * I
Set(name='({3, 4}*I)', dim=2, value=[(3, 1), (3, 2), (4, 1), (4, 2)])
>>> [3,4] * I
Set(name='([3, 4]*I)', dim=2, value=[(3, 1), (3, 2), (4, 1), (4, 2)])
```

pysimple.**__add__**(*obj1: ObjExp*, *obj2: ObjExp*) → *Parameter* | *Expression*

二項演算 obj1+obj2 を返します.

パラメータ

- **obj1** (*ObjExp*) --
- **obj2** (*ObjExp*) --

戻り値 obj1 または obj2 が Variable, Expression の場合は Expression を，それ以外の場合は Parameter を返します.

戻り値の型 *Parameter* or *Expression*

pysimple.**__sub__**(*obj1: ObjExp*, *obj2: ObjExp*) → *Parameter* | *Expression*

二項演算 obj1-obj2 を返します.

パラメータ

- **obj1** (*ObjExp*) --
- **obj2** (*ObjExp*) --

戻り値 obj1 または obj2 が Variable, Expression の場合は Expression を，それ以外の場合は Parameter を返します.

戻り値の型 *Parameter* or *Expression*

pysimple.**__mul__**(*obj1: ObjExp*, *obj2: ObjExp*) → *Parameter* | *Expression*

二項演算 obj1*obj2 を返します.

パラメータ

- **obj1** (*ObjExp*) --
- **obj2** (*ObjExp*) --

戻り値 obj1 または obj2 が Variable, Expression の場合は Expression を，それ以外の場合は Parameter を返します．演算結果が線形または二次でない場合，TypeError が投げられます.

戻り値の型 *Parameter* or *Expression*

例外 **TypeError** -- high dimension operand type(s) for *: {} and {}

pysimple.**__mod__**(*obj1: ObjPrm*, *obj2: ObjPrm*) → *Parameter*

二項演算 obj1%obj2 を返します.

パラメータ

- **obj1** (*ObjPrm*) --
- **obj2** (*ObjPrm*) --

戻り値 演算結果が線形または二次でない場合，TypeError が投げられます.

戻り値の型 *Parameter*

例外 **TypeError** -- high dimension operand type(s) for %: {} and {}

pysimple.**__truediv__**(*obj1: ObjExp*, *obj2: ObjPrm*) → *Parameter* | *Expression*

二項演算 obj1/obj2 を返します.

パラメータ

- **obj1** (*ObjExp*) --
- **obj2** (*ObjPrm*) --

戻り値 obj1 が Variable, Expression の場合は Expression を，それ以外の場合は Parameter を返します．演算結果が線形または二次でない場合，TypeError が投げられます.

戻り値の型 *Parameter* or *Expression*

例外 **TypeError** -- high dimension operand type(s) for /: {} and {}

pysimple.**__floordiv__**(*obj1: ObjPrm*, *obj2: ObjPrm*) → *Parameter*

二項演算 obj1//obj2 を返します.

パラメータ

- **obj1** (*ObjPrm*) --
- **obj2** (*ObjPrm*) --

戻り値 演算結果が線形または二次でない場合，TypeError が投げられます.

戻り値の型 *Parameter*

例外 **TypeError** -- high dimension operand type(s) for //: {} and {}

pysimple.**__pow__**(*obj1: ObjPrm*, *obj2: ObjPrm*) → *Parameter*

二項演算 obj1**obj2 を返します.

パラメータ

- **obj1** (*ObjPrm*) --
- **obj2** (*ObjPrm*) --

戻り値 演算結果が線形または二次でない場合，TypeError が投げられます.

戻り値の型 *Parameter*

例外 **TypeError** -- high dimension operand type(s) for **: {} and {}

pysimple.**__radd__**(*obj1: ObjExp*, *obj2: DType*) → *Parameter* | *Expression*

pysimple.**__rsub__**(*obj1: ObjExp*, *obj2: DType*) → *Parameter* | *Expression*

pysimple.**__rmul__**(*obj1: ObjExp*, *obj2: DType*) → *Parameter* | *Expression*

pysimple.**__rmod__**(*obj1: ObjPrm*, *obj2: DType*) → *Parameter*

pysimple.**__rdiv__**(*obj1: ObjPrm*, *obj2: DType*) → *Parameter*

pysimple.**__rtruediv__**(*obj1: ObjPrm*, *obj2: DType*) → *Parameter*

pysimple.**__rfloordiv__**(*obj1: ObjPrm*, *obj2: DType*) → *Parameter*

pysimple.**__rpow__**(*obj1: ObjPrm*, *obj2: DType*) → *Parameter*

同様に定義されています.

5.4.3 比較演算

pysimple.__lt__(*obj1: Iterable | ObjPrm*, *obj2: Iterable | ObjPrm*) → bool | *Cond*

比較演算 obj1<obj2 を返します．obj1, obj2 が共に collections.abc.Set の派生クラスの場合は集合の包含関係を，obj1, obj2 が共に ObjPrm の場合は不等式を満たす要素からなる条件文を，それ以外で ObjPrm と Iterable の比較は以下のような条件文を返します．

ObjPrm<Iterable の場合, set(ObjPrm) & set(Iterable), Iterable<ObjPrm の場合, set(ObjPrm) - set(Iterable) からなる条件文を返します．

パラメータ

- **obj1** (Iterable[*DType* | *Key*] or *ObjPrm*) --
- **obj2** (Iterable[*DType* | *Key*] or *ObjPrm*) --

戻り値 obj1, obj2 が共に collections.abc.Set の派生クラスの場合は bool，それ以外の場合は Cond を返します．ただし，両辺とも添字を含まない場合は bool を返します．

戻り値の型 bool or *Cond*

サンプル

```
>>> I = Set(value=[1,2,3,4], name='I')
>>> II = Set(value=[1,2,3,4,5], name='II')
>>> I < I
False
>>> I < II
True
>>> I < {1,2,3,4}
False
>>> i = Element(set=I, name='i')
>>> i < 3
(i<3)[i] in [1, 2]
>>> a = Parameter(index=i, name='a')
>>> a[i] = 4 - i
>>> a
a[1]=3
a[2]=2
a[3]=1
a[4]=0
>>> i < a[i]
(i<a[i])[i] in [1]
>>> J = Set(value=[2,4,6,8], name='J')
>>> i < J  # i.set & J
(i<J)[i] in [2, 4]
>>> J < i  # i.set - J
```

(次のページに続く)

(前のページからの続き)

```
(J<i)[i] in [1, 3]
>>> i < {2,4,6,8}
(i<{8, 2, 4, 6})[i] in [2, 4]
>>> i < [2,4,6,8]
(i<[2, 4, 6, 8])[i] in [2, 4]
```

pysimple.**__gt__**(*obj1: Iterable | ObjPrm*, *obj2: Iterable | ObjPrm*) → bool | *Cond*

比較演算 obj1>obj2 を返します．obj1, obj2 が共に collections.abc.Set の派生クラスの場合は集合の包含関係を，obj1, obj2 が共に ObjPrm の場合は不等式を満たす要素からなる条件文を，それ以外で ObjPrm と Iterable の比較は以下のような条件文を返します．

ObjPrm>Iterable の場合, set(ObjPrm) - set(Iterable), Iterable>ObjPrm の場合, set(ObjPrm) & set(Iterable) からなる条件文を返します．

パラメータ

- **obj1** (Iterable[*DType* | *Key*] or *ObjPrm*) --
- **obj2** (Iterable[*DType* | *Key*] or *ObjPrm*) --

戻り値 obj1, obj2 が共に collections.abc.Set の派生クラスの場合は bool，それ以外の場合は Cond を返します．ただし，両辺とも添字を含まない場合は bool を返します．

戻り値の型 bool or *Cond*

サンプル

```
>>> I = Set(value=[1,2,3,4], name='I')
>>> II = Set(value=[1,2,3], name='II')
>>> I > I
False
>>> I > II
True
>>> I > {1,2,3,4}
False
>>> i = Element(set=I, name='i')
>>> i > 3
(i>3)[i] in [4]
>>> a = Parameter(index=i, name='a')
>>> a[i] = 4 - i
>>> a
a[1]=3
a[2]=2
a[3]=1
a[4]=0
>>> i > a[i]
```

(次のページに続く)

(前のページからの続き)

```
(i>a[i])[i] in [3, 4]
>>> J = Set(value=[2,4,6,8], name='J')
>>> J > i  # i.set & J
(J>i)[i] in [2, 4]
>>> i > J  # i.set - J
(i>J)[i] in [1, 3]
>>> i > {2,4,6,8}
(i>{8, 2, 4, 6})[i] in [1, 3]
>>> i > [2,4,6,8]
(i>[2, 4, 6, 8])[i] in [1, 3]
```

pysimple.**__le__**(*obj1: Iterable | ObjExp*, *obj2: Iterable | ObjExp*) → bool | *Cond* | *Constraint*

比較演算 obj1<=obj2 を返します．obj1, obj2 が共に collections.abc.Set の派生クラスの場合は集合の包含関係を，obj1, obj2 が共に ObjPrm の場合は不等式を満たす要素からなる条件文を，obj1, obj2 に Variable, Expression を含む場合は制約式を返します．

パラメータ

- **obj1** (Iterable[*DType* | *Key*] or *ObjExp*) --
- **obj2** (Iterable[*DType* | *Key*] or *ObjExp*) --

戻り値 obj1, obj2 が共に collections.abc.Set の派生クラスの場合は bool，obj1, obj2 が共に ObjPrm の場合は Cond を，obj1, obj2 に Variable, Expression を含む場合は Expression を返します．ただし，obj1, obj2 が共に ObjPrm でいずれも添字含まない場合は bool を返します．

戻り値の型 bool or *Cond* or *Constraint*

サンプル

```
>>> I = Set(value=[1,2,3,4], name='I')
>>> II = Set(value=[1,2,3], name='II')
>>> I <= I
True
>>> I <= II
False
>>> I <= {1,2,3,4}
True
>>> i = Element(set=I, name='i')
>>> i <= 3
(i<=3)[i] in [1, 2, 3]
>>> a = Parameter(index=i, name='a')
>>> a[i] = 4 - i
>>> a
a[1]=3
```

(次のページに続く)

(前のページからの続き)

```
a[2]=2
a[3]=1
a[4]=0
>>> i <= a[i]
(i<=a[i])[i] in [1, 2]
>>> x = Variable(index=i, name='x')
>>> a[i] <= x[i]
(a[i]<=x[i]):
x[1]>=3
x[2]>=2
x[3]>=1
x[4]>=0
```

pysimple.__ge__(*obj1: Iterable | ObjExp*, *obj2: Iterable | ObjExp*) → bool | *Cond* | *Constraint*

比較演算 obj1>=obj2 を返します．obj1, obj2 が共に collections.abc.Set の派生クラスの場合は集合の包含関係を，obj1, obj2 が共に ObjPrm の場合は不等式を満たす要素からなる条件文を，obj1, obj2 に Variable, Expression を含む場合は制約式を返します．

パラメータ

- **obj1** (Iterable[*DType* | *Key*] or *ObjExp*) --
- **obj2** (Iterable[*DType* | *Key*] or *ObjExp*) --

戻り値 obj1, obj2 が共に collections.abc.Set の派生クラスの場合は bool，obj1, obj2 が共に ObjPrm の場合は Cond を，obj1, obj2 に Variable, Expression を含む場合は Expression を返します．ただし，obj1, obj2 が共に ObjPrm でいずれも添字含まない場合は bool を返します．

戻り値の型 bool or *Cond* or *Constraint*

サンプル

```
>>> I = Set(value=[1,2,3,4], name='I')
>>> II = Set(value=[1,2,3,4,5], name='II')
>>> I >= I
True
>>> I >= II
False
>>> I >= {1,2,3,4}
True
>>> i = Element(set=I, name='i')
>>> i >= 3
(i>=3)[i] in [3, 4]
>>> a = Parameter(index=i, name='a')
>>> a[i] = 4 - i
```

(次のページに続く)

(前のページからの続き)

```
>>> a
a[1]=3
a[2]=2
a[3]=1
a[4]=0
>>> i >= a[i]
(i>=a[i])[i] in [2, 3, 4]
>>> x = Variable(index=i, name='x')
>>> a[i] >= x[i]
(a[i]>=x[i]):
-x[1]>=-3
-x[2]>=-2
-x[3]>=-1
-x[4]>=0
```

pysimple.**__eq__**(*obj1: Iterable | ObjExp*, *obj2: Iterable | ObjExp*) → bool | *Cond* | *Constraint*

比較演算 obj1==obj2 を返します．obj1, obj2 が共に collections.abc.Set の派生クラスの場合は集合の包含関係を，obj1, obj2 が共に ObjPrm の場合は等式を満たす要素からなる条件文を，obj1, obj2 に Variable, Expression を含む場合は制約式を返します．

パラメータ

- **obj1** (Iterable[*DType* | *Key*] or *ObjExp*) --
- **obj2** (Iterable[*DType* | *Key*] or *ObjExp*) --

戻り値 obj1, obj2 が共に collections.abc.Set の派生クラスの場合は bool，obj1, obj2 が共に ObjPrm の場合は Cond を，obj1, obj2 に Variable, Expression を含む場合は Expression を返します．ただし，obj1, obj2 が共に ObjPrm でいずれも添字含まない場合は bool を返します．

戻り値の型 bool or *Cond* or *Constraint*

サンプル

```
>>> I = Set(value=[1,2,3,4], name='I')
>>> II = Set(value=[1,2,3], name='II')
>>> I == I
True
>>> I == II
False
>>> I == {1,2,3,4}
True
>>> i = Element(set=I, name='i')
>>> i == 3
(i==3)[i] in [3]
```

(次のページに続く)

(前のページからの続き)

```
>>> a = Parameter(index=i, name='a')
>>> a[i] = 4 - i
>>> a
a[1]=3
a[2]=2
a[3]=1
a[4]=0
>>> i == a[i]
(i==a[i])[i] in [2]
>>> x = Variable(index=i, name='x')
>>> a[i] == x[i]
(a[i]==x[i]):
-x[1]==-3
-x[2]==-2
-x[3]==-1
-x[4]==0
```

pysimple.**__ne__**(*obj1: collections.abc.Set | ObjPrm*, *obj2: collections.abc.Set | ObjPrm*) → bool | *Cond*

比較演算 obj1!=obj2 を返します．obj1, obj2 が共に collections.abc.Set の派生クラスの場合は集合の包含関係を，obj1, obj2 が共に ObjPrm の場合は不等式を満たす要素からなる条件文を返します．

パラメータ

- **obj1** (Set[*DType* | *Key*] or *ObjPrm*) --
- **obj2** (Set[*DType* | *Key*] or *ObjPrm*) --

戻り値 obj1, obj2 が共に collections.abc.Set の派生クラスの場合は bool，obj1, obj2 が共に ObjPrm の場合は Cond を返します．ただし，両辺とも添字を含まない場合は bool を返します．

戻り値の型 bool or *Cond*

サンプル

```
>>> I = Set(value=[1,2,3,4], name='I')
>>> II = Set(value=[1,2,3], name='II')
>>> I != I
False
>>> I != II
True
>>> I != {1,2,3,4}
False
>>> i = Element(set=I, name='i')
>>> i != 3
(i!=3)[i] in [1, 2, 4]
```

(次のページに続く)

(前のページからの続き)

```
>>> a = Parameter(index=i, name='a')
>>> a[i] = 4 - i
>>> a
a[1]=3
a[2]=2
a[3]=1
a[4]=0
>>> i != a[i]
(i!=a[i])[i] in [1, 3, 4]
```

5.4.4 ビット演算

Set.**__or__**(*rset: Iterable[DType | Key]*) → *Set*

Set と rset のいずれかに含まれる集合を返します.

パラメータ **rset** (Iterable[*DType* | *Key*]) -- Set や set でなくても構いません.

戻り値の型 *Set*

サンプル

```
>>> I = Set(value=[1,2,3,4], name='I')
>>> J = Set(value=[2,4,6,8], name='J')
>>> I | J
Set(name='(I|J)', value=[1, 2, 3, 4, 6, 8])
>>> I | {2,4,6,8}
Set(name='(I|{8, 2, 4, 6})', value=[1, 2, 3, 4, 8, 6])
>>> I | [2,4,6,8]
Set(name='(I|[2, 4, 6, 8])', value=[1, 2, 3, 4, 6, 8])
```

Set.**__and__**(*rset: Iterable[DType | Key]*) → *Set*

Set と rset の両方に含まれる集合を返します.

パラメータ **rset** (Iterable[*DType* | *Key*]) -- Set や set でなくても構いません.

戻り値の型 *Set*

サンプル

```
>>> I = Set(value=[1,2,3,4], name='I')
>>> J = Set(value=[2,4,6,8], name='J')
>>> I & J
Set(name='(I&J)', value=[2, 4])
>>> I & {2,4,6,8}
Set(name='(I&{8, 2, 4, 6})', value=[2, 4])
>>> I & [2,4,6,8]
Set(name='(I&[2, 4, 6, 8])', value=[2, 4])
```

Set.**__xor__**(*rset: Iterable[DType | Key]*) → *Set*

Set と rset のいずれか一方にだけ含まれる集合を返します.

パラメータ **rset** (Iterable[*DType* | *Key*]) -- Set や set でなくても構いません.

戻り値の型 *Set*

サンプル

```
>>> I = Set(value=[1,2,3,4], name='I')
>>> J = Set(value=[2,4,6,8], name='J')
>>> I ^ J
Set(name='(I^J)', value=[1, 3, 6, 8])
>>> I ^ {2,4,6,8}
Set(name='(I^{8, 2, 4, 6})', value=[1, 3, 8, 6])
>>> I ^ [2,4,6,8]
Set(name='(I^[2, 4, 6, 8])', value=[1, 3, 6, 8])
```

Set.**__ror__**(*lset: Iterable[DType | Key]*) → *Set*

lset と Set のいずれかに含まれる集合を返します.

パラメータ **lset** (Iterable[*DType* | *Key*]) -- Set や set でなくても構いません.

戻り値の型 *Set*

サンプル

```
>>> I = Set(value=[1,2,3,4], name='I')
>>> {2,4,6,8} | I
Set(name='({8, 2, 4, 6}|I)', value=[1, 2, 3, 4, 8, 6])
>>> [2,4,6,8] | I
Set(name='([2, 4, 6, 8]|I)', value=[1, 2, 3, 4, 6, 8])
```

Set.**__rand__**(*lset: Iterable[DType | Key]*) → *Set*

lset と Set の両方に含まれる集合を返します.

パラメータ **lset** (Iterable[*DType* | *Key*]) -- Set や set でなくても構いません.

戻り値の型 *Set*

サンプル

```
>>> I = Set(value=[1,2,3,4], name='I')
>>> {2,4,6,8} & I
Set(name='({8, 2, 4, 6}&I)', value=[2, 4])
>>> [2,4,6,8] & I
Set(name='([2, 4, 6, 8]&I)', value=[2, 4])
```

Set.**__rxor__**(*lset: Iterable[DType | Key]*) → *Set*

lset と Set のいずれか一方にだけ含まれる集合を返します.

パラメータ **lset** (Iterable[*DType* | *Key*]) -- Set や set でなくても構いません.

戻り値の型 *Set*

サンプル

```
>>> I = Set(value=[1,2,3,4], name='I')
>>> {2,4,6,8} ^ I
Set(name='({8, 2, 4, 6}^I)', value=[1, 3, 8, 6])
>>> [2,4,6,8] ^ I
Set(name='([2, 4, 6, 8]^I)', value=[1, 3, 6, 8])
```

Cond.**__or__**(*rcond:* Cond) → *Cond*

Cond と rcond のいずれかの条件を満たす条件式を返します.

パラメータ **rcond** (Cond) --

戻り値の型 *Cond*

サンプル

```
>>> i = Element(value=[1,2,3], name='i')
>>> (i<2) | (2<i)
((i<2)|(i>2))[i] in [1, 3]
```

注釈:

- 「演算子の優先順位」により，各条件には括弧をつける必要があります．

Cond.**__and__**(*rcond:* Cond) → *Cond*

Cond と rcond の両方の条件を満たす条件式を返します．

パラメータ **rcond** (Cond) --

戻り値の型 *Cond*

サンプル

```
>>> i = Element(value=[1,2,3], name='i')
>>> (1<i) & (i<3)
((i>1)&(i<3))[i] in [2]
```

注釈:

- 「演算子の優先順位」により，各条件には括弧をつける必要があります．
- 3 つ以上の条件文の場合，Condition 関数を使用した方が効率的で見やすいです．Condition(i, (1<i, i<5, i!=3))

参考:

pysimple.Condition

5.5 型ヒント

型ヒントで主に用いる PySIMPLE のクラスです．使用例については **型ヒントの使い方** をご覧ください．

5.5.1 PySIMPLE クラス

class pysimple.typing.**Set**

pysimple.Set と同じです．

class pysimple.typing.**Element**

pysimple.Element と同じです．

class pysimple.typing.**ElementSlice**

ElementSlice と同じです．コンストラクタは非公開ですが，型ヒントとして用いることができます．

class pysimple.typing.**Cond**

Cond と同じです．コンストラクタは非公開ですが，型ヒントとして用いることができます．

class pysimple.typing.**Parameter**

pysimple.Parameter と同じです．

class pysimple.typing.**Table**

Table と同じです．コンストラクタは非公開ですが，型ヒントとして用いることができます．

class pysimple.typing.**Variable**

pysimple.Variable と同じです．*pysimple.IntegerVariable* と *pysimple.BinaryVariable* インスタンスは Variable 型となることに注意ください．

class pysimple.typing.**Expression**

Expression と同じです．コンストラクタは非公開ですが，型ヒントとして用いることができます．

class pysimple.typing.**Constraint**

Constraint と同じです．コンストラクタは非公開ですが，型ヒントとして用いることができます．

class pysimple.typing.**SelectionConstraint**

SelectionConstraint と同じです．コンストラクタは非公開ですが，型ヒントとして用いることができます．*Constraint* の派生クラスです．

class pysimple.typing.**Index**

Index と同じです．コンストラクタは非公開ですが，型ヒントとして用いることができます．

5.5.2 基底クラス

pysimple.typing.**ELEMENT = <class 'pysimple.element.BaseElement'>**

Element, *ElementSlice*, *Cond* の抽象基底クラスです．

pysimple.typing.**PARAMETER = <class 'pysimple.table.BaseTable'>**

Parameter と *Table* の抽象基底クラスです．

5.5.3 エイリアス

Num

モデルに使用できる数値型です．int | float 型のエイリアスです．

DType

data type の意．キーや値として使用できる型です． *Num* | str 型のエイリアスです．

Key

集合オブジェクトの要素に使用できる型です．tuple[*DType*, ...] 型のエイリアスです．

pysimple.typing.**ObjPrm = int | float | str | pysimple.element.BaseElement | pysimple.table.BaseTable**

Represent a PEP 604 union type

E.g. for int | str

DType | *ELEMENT* | *PARAMETER* 型のエイリアスです.

pysimple.typing.**ObjExp: TypeAlias = int | float | str | pysimple.element.BaseElement | pysimple.table.BaseTable | pysimple.expression.Variable | pysimple.expression.Expression**

Represent a PEP 604 union type

E.g. for int | str

ObjPrm | *Variable* | *Expression* 型のエイリアスです.

第6章　更新履歴

6.1 [1.6.0] - 2025-03-25

Added

- **#12** Excel アドイン連携

Excel 上で Nuorium Optimizer の最適化モデルの入出力を行うことができる Excel アドインが PySIMPLE と連携できるようになりました.

- **#23** スタブファイル配布

スタブファイルが同梱されました．開発環境に依っては，シンタックスハイライト，メソッド・引数・型のサジェストが行われるようになります．また，型ヒントを記述しておくことで型チェッカーによる型チェックを補助できるようになります．型ヒントの詳細は **型ヒントの使い方** や **型ヒント** をご覧ください.

- **#24** 列生成法のサンプル

列生成法のサンプルを追加しました．詳細は **列生成法** をご覧ください.

- **#97** Python 3.13 対応

  Python 3.13 に対応しました.

- **#111** wls 並列化

  メタヒューリスティクスアルゴリズム wls が並列実行に対応しました．wls については **制約充足問題ソルバ** *wcsp/wls* をご覧ください．オプションの指定方法は **求解オプション** をご確認ください.

  Changed

- **#92** mpsout の引数

  `mpsout` の引数がキーワード引数専用となりました．この変更は後方互換性を損ないます．旧バージョンで `mpsout(filename)` と記述していた場合，`mpsout(mpsfile=filename)` と記述する必要があります.

  キーワード引数 anonymous と initial が追加されました．この追加は後方互換性を損ないます．旧バージョンと同じ挙動を期待する場合， `initial=True` と指定する必要があります.

  Removed

- **#73** 旧型式の求解オプション定数の廃止

  旧型式の求解オプション定数を削除しました.

```
>>> p = Problem()
>>> p.options.branchCut = Options.Branch.Cut.OFF  # 新形式 (1.4.0 or more)
>>> p.options.branchCut = Options.Branch.CUT_OFF  # 旧型式
AttributeError: type object 'Branch' has no attribute 'CUT_OFF'
>>> p.options.branchCut = 0  # 直接数値指定は警告
pysimple\options.py:224: FutureWarning: use Options.Branch.Cut.OFF instead of 0
>>>
>>> p.options.method = Options.Method.SIMPLEX
>>> p.options.method = 'simplex'  # method は直接数文字列指定でも警告とならない
```

- **#81** Python 3.9 サポート

  Python 3.9 は非対応となりました.

  Fixed

- **#1302** Sum の添字順を間違えることがある

  Sum 関数の引数が Parameter の場合に戻り値の添字順を間違うことがある不具合を修正しました.

6.2 [1.5.1] - 2024-07-02

Added

- **#94** NumPy 2 系対応

  メジャーバージョンアップした NumPy の 2 系に対応しました.

  Fixed

- **#99** wls の求解ステータスが正しくない

  wls の求解ステータス NuoptStatus の値が正しくないことがある不具合を修正しました.

- **#1256**

- **#1261**

- **#1278** mpsout で出力された mps ファイルのフォーマット

  mpsout で出力された mps ファイルのフォーマットに関する種々の不具合を修正しました.

6.3 [1.5.0] - 2024-03-28

Added

- **#72** 添字グループ情報

変数の添字グループ情報を指定できるようになりました．メタヒューリスティクスアルゴリズム wls へ，「割当てを意味する変数」の情報を明示的に渡せます．割当構造をもつ最適化問題に対し，探索性能の向上が見込めます．詳細は *Variable.group* をご覧ください．

- **#82** Python 3.12 対応

Python 3.12 に対応しました．

- **#85** 変数の値固定

変数の値を現在値に固定するメソッド *Variable.fix* を追加しました．また，固定した値を解除するメソッド *Variable.unfix* も利用できます．複数求解を行うときに，変数固定/解除を利用すると便利です．使用方法については **LP の値を丸めて MILP の近似解を得るテクニック** もご覧ください．

- **#89** 添字付き get メソッド

get メソッドを Parameter/Variable/Expression/Constraint に追加しました．get メソッドは __getitem__ の KeyError が投げられないバージョンで，key に対応するものがない場合には KeyError が投げられず 0 となります．これにより，境界条件やフローをより簡潔に記述することができるようになります．デフォルト値の指定はできず常に 0 となります．添字をサポートしないこれまでの Table.get メソッドは削除されました．詳細は *Parameter.get*, *Variable.get*, *Expression.get*, *Constraint.get* をご確認ください．使用方法については **境界条件やフロー保存則の記述テクニック** もご覧ください．

```
>>> i = Element(value=[1, 2, 3], name='i')
>>> z = BinaryVariable(index=i, name='z')
>>> z[i] - z.get(i-1) >= 1
((z[i]-z.get((i-1)[i])[i])[i]>=1):
z[1]>=1
-z[1]+z[2]>=1
-z[2]+z[3]>=1
```

- **#90** Parameter/Table の文字列フォーマット指定

添字なし Parameter/Table がフォーマット済み文字リテラル (f-string) に対応しました．目的関数値を `f'{problem.objective.val:.2f}'` のように成形するときなどに便利です．詳細は *Parameter.__format__* をご確認ください．

Changed

- **#88** wls の連続変数対応

メタヒューリスティクスアルゴリズム wls で **整数変数クラス** *IntegerVariable* だけでなく連続変数 **変数クラス** *Variable* を扱えるようになりました．連続変数を使う際は，目的関数と制約式が線形である必要があります．wls については **制約充足問題ソルバ** *wcsp/wls* をご覧ください．

Fixed

- **#1221** ネストした Sum でエラーになることがある

  ネストした Sum で添字が残っている場合に SimpleError になることがある不具合を修正しました.

6.4 [1.4.1] - 2023-06-15

Fixed

- **#1186** setitem で IndexError になることがある

  setitem で意図しない IndexError が発生することがある不具合を修正しました.

6.5 [1.4.0] - 2023-03-27

Added

- **#55** 分枝限定法の終了条件を関数で判定させる

  分枝限定法の終了条件を Python のコールバック関数で与えられるようになりました. 詳細は **コールバック関数** をご確認ください.

```
>>> def func(solver):
...     # 実行可能解の個数が 1 以上になったら分枝限定法を停止する
...     if solver['SolutionCount'] >= 1:
...         return True
...     return False
...
>>> problem = Problem()
>>> ...
>>> problem.setCallback(mip_terminate=func)
```

- **#63** Python 3.11 対応

  Python 3.11 に対応しました.

  Changed

- **#75** wcsp の一般整数変数対応

  制約充足問題に対するメタヒューリスティクスアルゴリズム wcsp で *0-1* **整数変数クラス** *BinaryVariable* だけでなく **整数変数クラス** *IntegerVariable* を使えるようになりました. wcsp については **制約充足問題ソルバ** *wcsp/wls* をご覧ください.

- **#70** NuoptStatus を列挙型にする

  NuoptStatus が IntEnum を継承するようになりました.

```
>>> x = BinaryVariable(name='x')
>>> p = Problem()
>>> p.status
<NuoptStatus.INITIAL: 0>
>>> print(p.status)
NuoptStatus.INITIAL
>>> p += x
>>> p.solve(silent=True)
<NuoptStatus.OPTIMAL: 1>
>>> p += x <= -1
>>> p.solve(silent=True)
<NuoptStatus.INFEASIBLE: 3>
```

- **#1113** 求解オプション定数を列挙型にする

  求解オプション定数が IntEnum を継承するようになりました．これに伴いクラスの構造が変わります．また，1.4.0 では新旧両方の指定方法で設定ができますが，将来的に従来の指定方法は廃止されます．値での指定は今後もサポートされます．

```
>>> p = Problem()
>>> p.options.branchCut
>>> p.options.branchCut = Options.Branch.CUT_AGGRESSIVE  # 従来の指定方法
>>> p.options.branchCut
2
>>> p.options.branchCut = Options.Branch.Cut.AGGRESSIVE  # バージョン 1.4.0 以降の
指定方法
>>> p.options.branchCut
<Cut.AGGRESSIVE: 2>
>>> p.options.branchCut = 2                              # 値で指定も可能 (従来通り)
>>> p.options.branchCut
2
```

- **#76** 選択型求解オプションの指定方式統一

  求解オプションの method と scaling で求解オプション定数が使用できるようになりました．method は今後とも新旧両方の指定方法がサポートされます．scaling は 1.4.0 では新旧両方の指定方法で設定ができますが，将来的に従来の文字列による指定方法は廃止されます．

```
>>> p = Problem()
>>> p.options.method = 'simplex'                 # 従来の指定方法
>>> p.options.method
'simplex'
>>> p.options.method = Options.Method.SIMPLEX  # バージョン 1.4.0 以降の指定方法
>>> p.options.method
<Method.SIMPLEX: 'simplex'>
>>> p.options.scaling = 'cr'                  # 従来の指定方法
>>> p.options.scaling
```

(次のページに続く)

(前のページからの続き)

```
<Scaling.CR: 2>
>>> p.options.scaling = Options.Scaling.CR  # バージョン 1.4.0 以降の指定方法
>>> p.options.scaling
<Scaling.CR: 2>
>>> p.options.scaling = 2                   # 値で指定も可能 (新規)
>>> p.options.scaling
2
```

Removed

- **#62** Python 3.8 サポート

Python 3.8 は非対応となりました.

Fixed

- **#50** LP の後に wcsp を解けない

連続変数を含む制約式・目的関数を一旦追加すると，最終的に使用していなくても wcsp として求解できなくなる不具合を修正しました.

6.6 [1.3.1] - 2022-06-30

Added

- **#6** Parameter と Set の比較

Parameter と Set の比較ができるようになりました.

```
>>> i = Element(value=[1, 2, 3], name='i')
>>> S = Set(value=[3, 4], name='S')
>>> i < S
(i<S)[i] in [3]
>>> i+1 < S
((i+1)[i]<S)[i] in [2, 3]
>>> i+1 < [3, 4]
((i+1)[i]<[3, 4])[i] in [2, 3]
>>> a = Parameter(index=i, name='a'); a[i]=i+1
>>> a[i] < S
(a[i]<S)[i] in [2, 3]
>>> a[i] < [3, 4]
(a[i]<[3, 4])[i] in [2, 3]
```

- **#8** Parameter を含んだ Expression のアクセス

Expression.__getitem__ に Parameter が使用できるようになりました.

```
>>> I = Set(value=[1,2]); i = Element(set=I, name='i')
>>> J = Set(value=[3,4]); j = Element(set=J, name='j')
>>> x = Variable(index=(i,j), init={ij: sum(ij) for ij in I*J}, name='x')
>>> e = x[i,j] + 1
>>> b = Parameter(index=j, value={3: 4, 4: 3}, name='b')
>>> e[i, b[j]].val
(x[i,j]+1)[i,b[j]][1,3].val=6
(x[i,j]+1)[i,b[j]][1,4].val=5
(x[i,j]+1)[i,b[j]][2,3].val=7
(x[i,j]+1)[i,b[j]][2,4].val=6
```

Fixed

- **#1022** 条件文の結果を間違うことがある

  修正漏れを再修正しました.

- **#1077** メモリリーク

  メモリリーク箇所の修正を行いました.

- **#1085** 実行可能解が見つかっていないのに isFeasible() が True

  実行可能解が得られていないにも関わらず *isFeasible* 関数が True になることがある不具合を修正しました.

6.7 [1.3.0] - 2022-03-28

Added

- **#4** IIS 情報の取得

  線形計画問題において互いに矛盾する制約式がある場合，その最小の組を IIS(Irreducible Infeasible Set) と呼びます．この IIS を Python オブジェクトとして取得できるようになりました．実行不可能な場合に，原因となる制約の特定や，その制約に対するケアなどを自動化する強力な手助けとなります．詳細は **実行不可能性要因検出機能** をご確認ください.

```
>>> x = Variable(name='x', lb=0)
>>> y = Variable(name='y', lb=0, ub=1)
>>> z = Variable(name='z', lb=0)
>>> p = Problem()
>>> p += x + y + z
>>> p += y + z >= 0, 'cons1'
>>> p += x + y >= 5, 'cons2'
>>> p += x + z <= 3, 'cons3'
>>> p.solve(silent=True)
3  # NuoptStatus.INFEASIBLE
>>> p.result.iis
```

(次のページに続く)

(前のページからの続き)

```
 0: cons2: violation=-1
     x+y>=5
 1: cons3
     -x-z>=-3
```

- **#42** PySIMPLE オブジェクトのシリアライズ

  PySIMPLE オブジェクトをバイナリにシリアライズできるようになりました．詳細は *PySIMPLE オブジェクトのシリアライズ* をご確認ください．

```
>>> i = Element(value=[1, 2, 3], name='i')
>>> x = Variable(index=i, name='x')
>>> with open('dump.pkl', 'wb') as f:
...     Serialize.dump(x, f)
...
>>> with open('dump.pkl', 'rb') as g:
...     x_ = Serialize.load(g)
...
```

- **#1036** Problem に登録されている変数一覧の取得

  Problem クラスに属性 variables が追加されました．詳細は *variables* をご確認ください．

- **#34** Problem の番号アクセス

  __getitem__ に番号アクセスできるようになりました．負数も使用可能です．*constraints* や *variables* も同様のアクセスができます．また，*__delitem__* においても同様です．

```
>>> x = Variable(name='x')
>>> y = Variable(name='y')
>>> p = Problem()
>>> p += x >= 1, 'cons1'
>>> p += y >= 1, 'cons2'
>>> p['cons2']
cons2:
y>=1
>>> p[1]
cons2:
y>=1
>>> p[-1]
cons2:
y>=1
>>> p.constraints['cons2']
cons2:
y>=1
>>> p.constraints[-1]
cons2:
```

(次のページに続く)

(前のページからの続き)

```
y>=1
>>> p.variables['y']
y:
y
>>> p.variables[-1]
y:
y
```

- **#35** 値の丸め

  組み込み関数 round でパラメータ値や計算結果の値を丸めることができるようになりました．詳細は *__round__* をご確認ください．

```
>>> p = Problem()
:
>>> p.solve(silent=True)
1
>>> x.val
x[0].val=1.500000000256067
x[1].val=2.9999999999710107
>>> round(x[i].val, 2)
round(x[i].val, 2)[0]=1.5
round(x[i].val, 2)[1]=3.0
```

- **#37** 目的関数の削除

  設定した目的関数を del 文で削除することができるようになりました．詳細は *objective* をご確認ください．

```
>>> x = Variable(name='x')
>>> y = Variable(name='y')
>>> p = Problem()
>>> p += x
>>> p.objective
x:
x
>>> p += y
pysimple.error.SimpleError: objective can only be assigned once
>>> del p.objective
>>> p += y
>>> p.objective
y:
y
```

- **#45** Python 3.10 対応

  Python 3.10 に対応しました．

Changed

- **#17** 制約種の LP/QP での利用

  制約式種 ハード制約，セミハード制約，ソフト制約を LP/QP の一部解法でも利用できるようになりました．

- **#33** 目的関数の初期値

  objective の初期値を空の式としました．以前は None でした．

```
>>> p = Problem()
>>> p.objective
<empty>:
0
```

- **#56** wcsp/wls で二次の制約

  wcsp/wls で二次の制約を扱えるようになりました．制約充足問題ソルバ *wcsp/wls*.

- **#57** wls で unbounded な整数変数

  wls で整数変数を扱う際に上下限制約がなくても扱えるようになりました．制約充足問題ソルバ *wcsp/wls*.

Removed

- **#46** Python 3.7 サポート

  Python 3.7 は非対応となりました．

6.8 [1.2.1] - 2021-06-16

Fixed

- **#919** KeyError のエラーメッセージ

  Expression 型に対する __getitem__ で発生する KeyError のメッセージを再考しました．メッセージは式の項単位となります．

```
>>> i = Element(value=[1,2], name='i')
>>> j = Element(value=[3,4], name='j')
>>> x = Variable(index=i, name='x')
>>> z = Variable(index=(i,j), name='z')
>>> exp = Sum(z[i,j], j) + x[i]
>>> exp[0]
KeyError: 'Sum(z[i,j], j)[0]'
```

- **#1004** mpsout で整数性が消失するケースがある

  変数が制約式にあらわれず，目的関数のみあるいは全くあらわれない場合， *mpsout* で出力される mps ファイルにおいて連続変数として記述されるという不具合を修正しました．

- **#1007** PySIMPLE の mpsout でファイル名を長くすると mpssolver で読めなくなる

  mpsout で指定されたファイル名が長すぎると，仕様を満たさない mps ファイルが出力されるという不具合を修正しました．

- **#1013** 範囲関数の空集合が含まれていたときの値

  PYSIMPLE-119 で演算結果に添字が残る場合のパラメータの結果を再修正しました．

```
>>> i0 = Element(value=[], name='i0')
>>> i1 = Element(value=[1,2], name='i1')
>>> a = Parameter(index=(i0,i1), name='a')
>>> a
# empty
>>> Sum(a[i0,i1], i0)  # fixed
Sum(a[i0,i1], i0)[1]=0
Sum(a[i0,i1], i0)[2]=0
>>> Sum(a[i0,i1], i1)
# empty
>>> Sum(a[i0,i1])
0
```

- **#1019** PySIMPLE の mpsout が目的関数の定数項に対応していない

  目的関数が定数項を含む場合，*mpsout* が目的関数を誤って出力する不具合を修正しました．

- **#1022** 条件文の結果を間違うことがある

  稀に条件文の結果を間違うことがある不具合を修正しました．

6.9 [1.2.0] - 2021-03-29

Added

- **PYSIMPLE-135** 二次計画問題対応

  制約式，目的関数に二次式を扱うことができるようになりました．

- **PYSIMPLE-136** wcsp/wls 対応

  重み付き制約充足問題に対するメタヒューリスティクス解法 wcsp を呼び出せるようになりました．新たなメタヒューリスティクスアルゴリズム wls を利用できるようになりました．

- **PYSIMPLE-172** Python 3.9 対応

  Python 3.9 に対応しました．

Changed

- **PYSIMPLE-185** Variable/Expression.val の戻り値型の統一性

  添字を持たない式に対する `.val` 等の属性は *Table* を返すようにしました．以前は組込み型でした．特に目的関数が該当するので注意ください．

- **PYSIMPLE-192** 変数の bound error のタイミング

  変数の bound チェックを行うタイミングを求解時から宣言時にしました．ただし，*type* 属性を後から変更したタイミングでは bound チェックは行われません．

```
>>> x = Variable(lb=3, ub=2, name='x')
pysimple.error.SimpleError: infeasible bound of variable x ( defined infeasible␣
↪bound [3 <= * <= 2] )
```

Removed

- **PYSIMPLE-198** Python 3.6 サポート

  Python 3.6 は非対応となりました．

Fixed

- **#967** 同類項が整理されていない式の二項演算の結果を間違う

  同類項が整理されていない式に対して二項演算の結果を間違う不具合を修正しました．同類項の整理アルゴリズムの強化に加え，同類項が整理されていない場合でも問題ないアルゴリズムとしました．

- **#992** setitem でエラーになることがある

  setitem で意図しない IndexError が発生することがある不具合を修正しました．

- **PYSIMPLE-118** 条件を更に Condition で条件付けると SimpleError になる

  Cond 型のオブジェクトを Condition に用いると SimpleError になる不具合を修正しました．

```
>>> i = Element(value=[0,1,2], name='i')
>>> i0 = i>0
>>> Condition(i0, i0<2)
((i>0), ((i>0)<2))[(i>0)] in [1]
```

6.10 [1.1.2] - 2020-09-11

Fixed

- **#954** 変数の同類項を複数含む式同士の加減でエラー

  計算でエラーとなっていたものを修正しました．

```
>>> i = Element(value=[1,2], name='i')
>>> x = Variable(index=i, name='x')
>>> y = Variable(index=i, name='y')
>>> e = x[i] + y[i]
>>> e
(x[i]+y[i]):
x[1]+y[1]
x[2]+y[2]
```

(次のページに続く)

(前のページからの続き)

```
>>> e + e  # fixed
((x[i]+y[i])[i]+(x[i]+y[i])[i]):
2*x[1]+2*y[1]
2*x[2]+2*y[2]
```

- **#965** 上下限値や定数項に 2^31 以上の整数型を指定するとオーバーフロー

  変数の上下限値や定数項に 2^31 以上の整数型を指定すると求解時に OverflowError となる不具合を修正しました.

6.11 [1.1.1] - 2020-06-08

Added

- **#883** Collection メソッドのサポート

  collections.abc.Collection がサポートするメンバ __contains__, __len__, __iter__ を *Parameter*, *Table*, *Variable*, *Expression*, *Constraint*, のすべてがサポートするようになりました.

Fixed

- **#933** 添字が逆転する演算を間違う

  `j*x[i,j]` のように，変数の添字に対して演算結果の添字が逆転する場合に計算結果を誤る不具合を修正しました.

```
>>> i = Element(value='XY', name='i')
>>> j = Element(value=[1, 2], name='j')
>>> x = Variable(index=(i,j), name='x')
>>> j*x[i,j]  # Index(j,i)
(j*x[i,j]):
x['X',1]
x['Y',1]
2*x['X',2]
2*x['Y',2]
```

- **PYSIMPLE-181** 範囲関数の範囲重複判定の厳密化

  範囲関数の範囲に同じ添字を含んでいてもスライス単位で重複していなければ許可されるようになりました.

- **PYSIMPLE-184** 特定の式に対する二項演算ができない

  特定の変数・式同士の二項演算ができない不具合を修正しました.

6.12 [1.1.0] - 2020-03-02

PySIMPLE 1.1.0 では式展開が劇的に高速化されました．1.0.1 と比較して 10 倍以上の高速化を達成し，C++SIMPLE と比較しても遜色のない速度になりました．

また，Nuorium 統合環境 も外部コマンドの実行に対応し，Nuorium 統合環境上で PySIMPLE が動作するようになりました．→ **数理計画問題を解く**

動作環境も Linux，Mac がサポートされるようになり，最新の Python 3.8 にも対応しました．

Added

- **PYSIMPLE-140** Linux，Mac 版

  Linux，Mac 版に対応しました．

- **PYSIMPLE-158** Python 3.8 対応

  Python 3.8 に対応しました．

- **PYSIMPLE-157** Set のスライス

  __getitem__ の引数にスライスオブジェクトが対応しました．

```
>>> I = Set(value=[1,2,3], name='I')
>>> I
Set(name='I', value=[1, 2, 3])
>>> I[:-1]
Set(name='I[:-1]', value=[1, 2])
```

- **#839** 求解オプションの初期化

  設定した求解オプションを del 文で初期値に戻すことができるようになりました．

```
>>> p = Problem()
>>> print(p.options.maxTime)
None
>>> p.options.maxTime = 1
>>> p.options.maxTime
1
>>> del p.options.maxTime
>>> print(p.options.maxTime)
None
```

  また，`del p.options` ですべての求解オプションを初期値に戻すことができます．詳細は **求解オプション** をご確認ください．

- **PYSIMPLE-80** mpsout

  Problem に mpsout メソッドが追加されました．詳細は *mpsout* をご確認ください．

Changed

- **PYSIMPLE-153** bound 周りの仕様

  lb, *ub* 属性は immutable になり，bound 制約は制約式に一本化されました．これにより変数，制約式の dual が正しく取得できます．

```
>>> i = Element(value=[1,2], name='i')
>>> x = Variable(index=i, lb=15, name='x')
>>> x.lb
x[1].lb=15
x[2].lb=15
>>> p = Problem()
>>> p += x[i] >= i*10, 'cons'
>>> x.lb
x[1].lb=15
x[2].lb=15
>>> p += Sum(x[i])
>>> p.solve(silent=True)
1
>>> x.val
x[1].val=15.000000041666626
x[2].val=20.000000041666624
>>> x.dual
x[1].dual=1.0
x[2].dual=0.0
>>> p['cons'].dual
cons[1].dual=0.0
cons[2].dual=1.0
```

Removed

- **PYSIMPLE-167** Python 3.5 サポート

  Python 3.5 は非対応となりました．

Fixed

- **#805** 多次元集合の一部メソッドの引数制限

  実装の都合により，多次元集合に対する *__getitem__*, *next*, *prev* の引数に *ELEMENT* を禁止します．*index* では可能です．

```
>>> IJ = Set(value=[(1,2), (3,4), (5,6)], name='IJ')
>>> i = Element(value=[0,2], name='i')
>>> IJ[0]
(1, 2)
>>> IJ[i]
TypeError: cannot apply IJ[i] to multidimensional Set
>>> ij = Element(value=[(3,4)], name='ij')
>>> IJ.next((3,4))
```

(次のページに続く)

(前のページからの続き)

```
(5, 6)
>>> IJ.next(ij)
TypeError: cannot apply IJ.next(ij) to multidimensional Set
>>> IJ.index((3,4))
1
>>> IJ.index(ij)
IJ.index(ij)[3,4]=1
```

- **#852** Parameter * Variable/Expression の添字順

  Parameter * Variable/Expression の演算結果の添字順を出現順にしました.

```
>>> i = Element(value=[1,2], name='i')
>>> j = Element(value=[3,4], name='j')
>>> a = Parameter(index=(i,j), name='a')
>>> x = Variable(index=j, name='x')
>>> a[i,j] = 10*i+j
>>> ax = a[i,j]*x[j]
>>> ax
(a[i,j]*x[j]):
13*x[3]
14*x[4]
23*x[3]
24*x[4]
>>> ax.index
Index(i,j)
```

- **PYSIMPLE-69** 目的関数の定数値の処理

  目的関数に定数項が存在した場合，求解情報や *optValue* に反映されない不具合を修正しました. *objective* の val 属性は今まで通り正しい値が入ります.

```
>>> i = Element(value=[1,2])
>>> x = BinaryVariable(index=i)
>>> p = Problem(type=max)
>>> p += Sum(x[i]) + 10
>>> p.solve()
 :
VALUE_OF_OBJECTIVE                                    12
 :
>>> p.result.optValue
12.0
>>> p.objective.val
12
```

- **PYSIMPLE-76** IPython 環境から実行した場合の出力先

求解情報がコンソールに出力される不具合を修正しました.

- **PYSIMPLE-137** silent=True で主要なインタプリタから求解できない

  IDLE や IPython などのインタプリタ上で `silent=True` で求解できない不具合を修正しました.

6.13 [1.0.1] - 2019-07-01

Fixed

- **PYSIMPLE-104** 目的関数に与えていない変数の初期値が変わる

  一部を目的関数に与えた変数のうち，与えていない部分の値が変わる不具合を修正しました.

```
>>> p = Problem()
>>> i = Element(value=[1,2], name='i')
>>> x = Variable(index=i, lb=1, init=2, name='x')
>>> x.val
x[1].val=2
x[2].val=2
>>> p += x[2]
>>> p.solve(silent=True)
1
>>> x.val
x[1].val=2  # 2 のままになるようにした
x[2].val=1.0000000002083331
```

- **PYSIMPLE-117** 範囲演算における戻り値の添字順

  範囲演算における戻り値の添字順を出現順にしました.

```
>>> ij = Element(value=[(1,3), (1,4), (2,3)], name='ij')
>>> i = Element(set=ij.set(0), name='i')
>>> j = Element(set=ij.set(1), name='j')
>>> y = Variable(index=(i,j), name='y')
>>> y[i,j] = 10*i+j
>>> y.val
y[1,3].val=13
y[1,4].val=14
y[2,3].val=23
y[2,4].val=24
>>> Sum(y[ij(0),j], ij(0)).index
Index(ij(1),j)  # 以前は (j,ij(1)) だった
>>> Sum(y[ij(0),j], ij(0))
Sum(y[ij(0),j], ij(0)):
y[1,3]+y[2,3]
y[1,4]+y[2,4]
```

(次のページに続く)

(前のページからの続き)

```
y[1,3]
y[1,4]
>>> Sum(y[ij(0),j], ij(0)).val
Sum(y[ij(0),j], ij(0))[3,3].val=36
Sum(y[ij(0),j], ij(0))[3,4].val=38
Sum(y[ij(0),j], ij(0))[4,3].val=13
Sum(y[ij(0),j], ij(0))[4,4].val=14
```

- **PYSIMPLE-119** 範囲関数の範囲に空集合が含まれていたときの値

  挙動を C++SIMPLE に合わせました.

```
>>> i0 = Element(value=[], name='i0')
>>> i1 = Element(value=[1], name='i1')
>>> y = Variable(index=(i0,i1), name='y')
>>> Sum(y[i0,i1], i0)
Sum(y[i0,i1], i0):
0
>>> Sum(y[i0,i1], i1)
Sum(y[i0,i1], i1):
# 空の式が返るようにした
>>> Sum(y[i0,i1], (i0,i1))
Sum(y[i0,i1], (i0,i1)):
0
```

- **PYSIMPLE-125** Parameter を含むアクセスをした式の表示

  少し複雑なアクセスを含む式の表示がおかしくなる不具合を修正しました.

```
>>> i = Element(value=[1,2,3], name='i')
>>> x = Variable(index=i, name='x')
>>> t = Element(value=[1,2], name='t')
>>> x[t]
x:
x[1]
x[2]
>>> x[t+1]
x[(t+1)[t]]:
x[(t+1)[t]][1]
x[(t+1)[t]][2]
>>> x[t+1]+1
(x[(t+1)[t]][t]+1):
x[2]+1
x[3]+1
>>> x[t] + x[t+1]
(x[t]+x[(t+1)[t]][t]):
```

(次のページに続く)

(前のページからの続き)

```
x[1]+x[2]
x[2]+x[3]
>>> x[t+1] + x[t]
(x[(t+1)[t]][t]+x[t]):
x[1]+x[2]
x[2]+x[3]
```

6.14 [1.0.0] - 2019-03-08

リリース

第7章　ライセンス

END-USER LICENSE AGREEMENT FOR NTT DATA MATHEMATICAL SYSTEMS INC. IMPORTANT PLEASE READ THE TERMS AND CONDITIONS OF THIS LICENSE AGREEMENT CAREFULLY BEFORE CONTINUING WITH THIS PROGRAM INSTALL: NTT DATA MATHEMATICAL SYSTEMS INC.'s End-User License Agreement ("EULA") is a legal agreement between you (either an individual or a single entity) and NTT DATA Mathematical Systems Inc. for the NTT DATA Mathematical Systems Inc. software product(s) which may include associated software components, media, printed materials, and "online" or electronic documentation ("PySIMPLE"). By installing, copying, or otherwise using PySIMPLE, you agree to be bound by the terms of this EULA. This license agreement represents the entire agreement concerning the program between you and NTT DATA Mathematical Systems Inc., (referred to as "licenser"), and it supersedes any prior proposal, representation, or understanding between the parties. If you do not agree to the terms of this EULA, do not install or use PySIMPLE.

PySIMPLE is protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. The PySIMPLE is licensed, not sold.

1. GRANT OF LICENSE.

PySIMPLE is licensed as follows:

(a) Installation and Use. NTT DATA Mathematical Systems Inc. grants you the right to install and use copies of PySIMPLE on your computer running a validly licensed copy of the operating system for which PySIMPLE was designed.

(b) Backup Copies. You may also make copies of PySIMPLE as may be necessary for backup and archival purposes.

2. DESCRIPTION OF OTHER RIGHTS AND LIMITATIONS.

(a) Maintenance of Copyright Notices. You must not remove or alter any copyright notices on any and all copies of PySIMPLE.

(b) Distribution. You may not distribute registered copies of PySIMPLE to third parties.

(c) Prohibition on Reverse Engineering, Decompilation, and Disassembly. You may not reverse engineer, decompile, or disassemble PySIMPLE, except and only to the extent that such activity is expressly permitted by applicable law notwithstanding this limitation.

(d) Rental. You may not rent, lease, or lend PySIMPLE.

(e) Support Services. NTT DATA Mathematical Systems Inc. may provide you with support services related to PySIMPLE ("Support Services"). Any supplemental software code provided to you as part of the Support Services shall be considered part of PySIMPLE and subject to the terms and conditions of this EULA.

(f) Compliance with Applicable Laws. You must comply with all applicable laws regarding use of PySIMPLE.

3. TERMINATION

Without prejudice to any other rights, NTT DATA Mathematical Systems Inc. may terminate this EULA if you fail to comply with the terms and conditions of this EULA. In such event, you must destroy all copies of PySIMPLE in your possession.

4. COPYRIGHT

All title, including but not limited to copyrights, in and to PySIMPLE and any copies thereof are owned by NTT DATA Mathematical Systems Inc. All title and intellectual property rights in and to the content which may be accessed through use of PySIMPLE is the property of the respective content owner and may be protected by applicable copyright or other intellectual property laws and treaties. This EULA grants you no rights to use such content. All rights not expressly granted are reserved by NTT DATA Mathematical Systems Inc.

5. NO WARRANTIES

NTT DATA Mathematical Systems Inc. expressly disclaims any warranty for PySIMPLE. The PySIMPLE is provided 'As Is' without any express or implied warranty of any kind, including but not limited to any warranties of merchantability, noninfringement, or fitness of a particular purpose. NTT DATA Mathematical Systems Inc. does not warrant or assume responsibility for the accuracy or completeness of any information, text, graphics, links or other items contained within PySIMPLE. NTT DATA Mathematical Systems Inc. makes no warranties respecting any harm that may be caused by the transmission of a computer virus, worm, time bomb, logic bomb, or other such computer program. NTT DATA Mathematical Systems Inc. further expressly disclaims any warranty or representation to Authorized Users or to any third party.

6. LIMITATION OF LIABILITY

In no event shall NTT DATA Mathematical Systems Inc. be liable for any damages (including, without limitation, lost profits, business interruption, or lost information) rising out of 'Authorized Users' use of or inability to use PySIMPLE, even if NTT DATA Mathematical Systems Inc. has been advised of the possibility of such damages. In no event will NTT DATA Mathematical Systems Inc. be liable for loss of data or for indirect, special, incidental, consequential (including lost profit), or other damages based in contract, tort or otherwise. NTT DATA Mathematical Systems Inc. shall have no liability with respect to the content of PySIMPLE or any part thereof, including but not limited to errors or omissions contained therein, libel, infringements of rights of publicity, privacy, trademark rights, business interruption, personal injury, loss of privacy, moral rights or the disclosure of confidential information.

NTT DATA Mathematical Systems Inc. Rengakan 1F 35 SHINANOMACHI, SHINJUKUKU, TOKYO, 160-0016 JAPAN

Python モジュール索引

p

s

索引

記号

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

モ