MoselPy

User Guide and Reference Manual

6.12

Beta release version

FICO® Xpress Optimization

FICO

©2026 Fair Isaac Corporation. All rights reserved. This documentation is the property of Fair Isaac Corporation (”FICO”). Receipt or possession of this documentation does not convey rights to disclose, reproduce, make derivative works, use, or allow others to use it except solely for internal evaluation purposes to determine whether to purchase a license to the software described in this documentation, or as otherwise set forth in a written software license agreement between you and FICO (or a FICO affiliate). Use of this documentation and the software described in it must conform strictly to the foregoing permitted uses, and no other use is permitted.

The information in this documentation is subject to change without notice. If you find any problems in this documentation, please report them to us in writing. Neither FICO nor its affiliates warrant that this documentation is error-free, nor are there any other warranties with respect to the documentation except as may be provided in the license agreement. FICO and its affiliates specifically disclaim any warranties, express or implied, including, but not limited to, non-infringement, merchantability and fitness for a particular purpose. Portions of this documentation and the software described in it may contain copyright of various authors and may be licensed under certain third-party licenses identified in the software, documentation, or both.

In no event shall FICO or its affiliates be liable to any person for direct, indirect, special, incidental, or consequential damages, including lost profits, arising out of the use of this documentation or the software described in it, even if FICO or its affiliates have been advised of the possibility of such damage. FICO and its affiliates have no obligation to provide maintenance, support, updates, enhancements, or modifications except as required to licensed users under a license agreement.

FICO is a registered trademark of Fair Isaac Corporation in the United States and may be a registered trademark of Fair Isaac Corporation in other countries. Other product and company names herein may be trademarks of their respective owners.

Patent(s): www.fico.com/en/patents

FICO® Xpress Mosel 6.12 (FICO® Xpress 9.9)

Deliverable Version: A

Last Revised: 15 May, 2026

Contents

Overview 1

Document scope . . . 1
Relationship to python3.dso . . . 2

I User Guide 3

1 Getting Started with MoselPy 4

1.1 Introduction . . . 4
1.1.1 When to use MoselPy . . . 4
1.1.2 Prerequisites . . . 4
1.2 Installation . . . 5
1.2.1 Environment variables . . . 5
1.3 Your first model . . . 5
1.3.1 The three-step workflow . . . 6
1.3.2 Model parameters . . . 6
1.4 Stream redirection . . . 7
1.4.1 Silencing output . . . 7
1.4.2 Capturing output . . . 7
1.4.3 Global vs per-model redirection . . . 7
1.5 Running a model from a string . . . 7

2 Data Exchange 9

2.1 Python to Mosel data flow . . . 9
2.1.1 Basic pattern . . . 9
2.2 Type mapping . . . 10
2.2.1 Scalars . . . 10
2.2.2 Arrays . . . 10
2.2.3 Sets . . . 11
2.2.4 Lists . . . 11
2.3 Pandas integration . . . 11
2.4 Parameterising the data source . . . 12

3 Optimization Workflows 13

3.1 Running optimization models . . . 13
3.1.1 Mosel model . . . 13
3.1.2 Python code . . . 14
3.2 Problem status . . . 15
3.3 Accessing results . . . 15
3.3.1 Method 1: run() return value . . . 16
3.3.2 Method 2: find_identifier() . . . 16
3.4 Visualizing results . . . 17

3.4.1 Retrieving results as Pandas objects . . . 17
3.4.2 Production dashboard . . . 17

4 Advanced Topics 19

4.1 Iterative solving . . . 19
4.1.1 Capacity scenario analysis . . . 19
4.1.2 Plotting scenario results . . . 20
4.2 Model control . . . 20
4.2.1 Graceful stop . . . 20
4.2.2 Immediate termination . . . 21
4.3 Model introspection . . . 21
4.4 Annotations . . . 21
4.5 Context managers . . . 22
4.6 Performance considerations . . . 22
4.6.1 Binary format for large data . . . 22
4.7 Exporting problems . . . 23

II API Reference 24

5 Configuration 25

Config . . . 26
5.1 Config properties . . . 27
bim_prefix . . . 27
default_working_directory . . . 27
locale_directory . . . 27
max_stack_dump_depth . . . 27
Config.set_default_stream . . . 29
5.2 Package attributes . . . 30
bim_version . . . 30
config . . . 30
library_path . . . 30
mosel_version . . . 30
native_version . . . 30
symbols . . . 31
version . . . 31
xpress_version . . . 31

6 Model Class 32

Model . . . 33
Model.run . . . 34
Model.find_identifier . . . 35
Model.stop . . . 36
Model.terminate . . . 37
Model.reset . . . 38
Model.export_problem . . . 39
Model.get_annotations . . . 40
Model.name . . . 41
Model.number . . . 42
Model.is_loaded . . . 43
Model.is_running . . . 44
Model.exec_status . . . 45

Model.exit_code . . . 46
Model.objective_value . . . 47
Model.prob_status . . . 48
Model.sec_status . . . 49
Model.key_fingerprint . . . 50
Model.size . . . 51
Model.version . . . 52
Model.symbols . . . 53
Model.sys_comment . . . 54
Model.user_comment . . . 55
Model.set_default_stream . . . 56

7 Package Functions 57

compile_model . . . 58
load_model . . . 59
execute_model . . . 60
remove_temp_directory . . . 61

8 I/O Driver 62

8.1 Driver *moselpy* without a label . . . 62
8.1.1 Type conversion . . . 63
8.1.2 Sparse arrays . . . 63
8.2 Driver *moselpy* with a label . . . 64

9 Status Classes 65

9.1 ProblemStatus class . . . 65
ProblemStatus . . . 66
ProblemStatus.status_code . . . 67
ProblemStatus.is_problem_valid . . . 68
ProblemStatus.is_solution_available . . . 69
ProblemStatus.is_solution_optimal . . . 70
ProblemStatus.is_optimization_finished . . . 71
ProblemStatus.is_problem_infeasible . . . 72
ProblemStatus.is_problem_unbounded . . . 73
ProblemStatus.is_optimization_failed . . . 74
9.2 SecurityStatus class . . . 75
SecurityStatus . . . 76
SecurityStatus.status_code . . . 77
SecurityStatus.is_signed . . . 78
SecurityStatus.is_encrypted . . . 79
SecurityStatus.is_signature_checked . . . 80
SecurityStatus.is_signature_verified . . . 81

10 Enumerations 82

10.1 ExecStatus enumeration . . . 82
ExecStatus . . . 83
10.2 StreamType enumeration . . . 84
StreamType . . . 85

11 Binary Format Utilities 86

to_binary . . . 87

from_binary 88
write_binary 89
read_binary 90

Appendix 91

A Contacting FICO 91

FICO Customer Support 91
Documentation 91
FICO Learning 92
Sales and maintenance 92
About FICO 92

Index 93

Overview

MoselPy is a Python package that provides bindings to FICO Xpress Mosel, enabling seamless integration between Python and Mosel. It allows Python developers to:

- compile and run Mosel models from Python,
- exchange data bidirectionally (Python $\leftrightarrow$ Mosel),
- integrate with Pandas Series and DataFrames,
- control Mosel execution and stream redirection,
- access optimization results programmatically,
- query model status, security, and annotations.

A typical workflow involves three steps: prepare your data in Python, run a Mosel model to solve an optimization problem, and retrieve the results back in Python. The *moselpy* I/O driver for data exchange in memory handles data conversion automatically.

Document scope

This document serves as both a tutorial introduction and a complete API reference for the *moselpy* package.

Part 1: User Guide introduces MoselPy through practical examples, focusing on the core workflows needed to integrate Mosel optimization models into Python applications and data science workflows.

- Chapter 1 covers installation, prerequisites, and your first MoselPy model, introducing the basic compile-load-run workflow.
- Chapter 2 explains bidirectional data exchange between Python and Mosel, including type mapping, arrays, sets, and Pandas integration.
- Chapter 3 demonstrates running optimization models, accessing solution values, checking problem status, and visualizing results with matplotlib.
- Chapter 4 covers advanced topics including iterative solving, model control, annotations, security, binary format, and stream redirection.

Part 2: API Reference provides a complete reference for all classes, functions, enumerations, and the *moselpy* I/O driver.

Relationship to python3.dso

The *moselpy* package and the *python3* Mosel module address opposite use cases:

Component	Host Language	Called Language	Use Case
`python3.dso`	Mosel	Python	Use Python libraries from Mosel models
`moselpy`	Python	Mosel	Embed Mosel models in Python applications

Choose *moselpy* when your main application is written in Python and you want to call Mosel code, in particular for optimization. Choose *python3.dso* when your main application is a Mosel model and you want to leverage advanced Python functionality such as specialized analytics, statistics, or graphical libraries. Using both components together (i.e., a Mosel model that uses *python3.dso* running inside a *moselpy* host) is not currently supported.

I. User Guide

Chapter 1

Getting Started with MoselPy

1.1 Introduction

MoselPy combines Python's ecosystem with Mosel's optimization capabilities, leveraging Mosel's performance, portability, and security characteristics while enabling Python-based data preparation, result analysis, and workflow integration.

1.1.1 When to use MoselPy

Use MoselPy when you:

- have existing Mosel models and want to integrate them into Python applications
- prefer Mosel's algebraic modeling syntax, which closely mimics mathematical notation, for clarity and maintainability
- need pre-compiled models for performance optimization or intellectual property protection
- want to leverage Mosel's efficient integration with Xpress Solver while using Python for data preparation
- require seamless Pandas Series and DataFrames integration for input/output in data science workflows
- need iterative solving with Python-based logic between solves, combining Mosel's solving power with Python's flexibility
- are building data-driven optimization workflows that benefit from Mosel's efficiency and dynamic object support.

1.1.2 Prerequisites

MoselPy requires:

- Python 3.10 to 3.14 (standard builds). Free-threaded Python (`3.13t`, `3.14t`) is not currently supported.
- FICO Xpress installation (a solver license is only required to run optimization models; the Mosel runtime itself is license-free). The Xpress Solver libraries are only required when the Mosel model uses `mmxprs` or `mmxnlp`.
- Optional: Pandas (for Series and DataFrame integration)

1.2 Installation

Install MoselPy via PyPI:

```
pip install moselpy
```

Verify installation:

```
import moselpy as mp
print(f"MoselPy version: {mp.version}")
print(f"Mosel version: {mp.mosel_version}")
print(f"Xpress version: {mp.xpress_version}")
```

The package includes the Python bindings; the Mosel runtime and Xpress Optimizer libraries are provided by your Xpress installation (see the Xpress Installation Guide).

Runtime dependencies. MoselPy requires `xprm_rt` (Mosel runtime) and `xprm_mc` (Mosel compiler) from your Xpress installation. If a Mosel model loads Mosel libraries (BIM or DSO files) these also need to be present with their respective dependencies as indicated in the **??** Xpress Installation Guide.

1.2.1 Environment variables

MoselPy uses the standard Xpress environment variables, which are automatically set up by the Xpress installers. The following are relevant when running MoselPy:

`XPRESSDIR` Path to the Xpress installation directory. Required if Xpress is not installed in the default location.

`XPAUTH_PATH` Path to the solver license file (`xpauth.xpr`). Searched in standard locations if not specified. Only needed for solver-based models.

Note on Jupyter notebooks. Jupyter spawns a separate kernel process that may not inherit the full system environment. This causes solver modules such as `mmxprs` to be not found even when `XPRESSDIR` is correctly set. If you encounter this, add the following line at the top of your notebook setup cell, before importing `moselpy`:

```
import os
os.environ.setdefault('XPRESSDIR', '/path/to/xpressmp')
```

1.3 Your first model

We start with a simple Mosel model that prints a message. Create a file `hello.mos`:

```
model HelloWorld
  writeln("Hello from Mosel!")
end-model
```

Execute it from Python:

```
import moselpy as mp

# One-step: compile, load, and run
mp.execute_model("hello.mos")
```

Output: `Hello from Mosel!`

1.3.1 The three-step workflow

For more control, use the three-step workflow: compile, load, run. The key functions are `compile_model()` to generate a compiled BIM file, `load_model()` to load it into memory, and `run()` to execute the model:

```
import moselpy as mp

# 1. Compile: .mos -> .bim
mp.compile_model("hello.mos", "hello.bim")

# 2. Load compiled model
model = mp.load_model("hello.bim")

# 3. Run
model.run()

print(f"Model '{model.name}' completed with status: {model.exec_status}")
```

Output:

```
Hello from Mosel!
Model 'HelloWorld' completed with status: ExecStatus.OK
```

The compiled `.bim` file is platform-independent and can be reused without recompilation, providing *faster execution* and *intellectual property protection* by hiding the model source code.

1.3.2 Model parameters

Pass parameters to Mosel models at execution time using the `exec_params` argument in the `run()` method. Parameters defined in the Mosel `parameters` block can be overridden from Python. Create `greeting.mos`:

```
model Greeting
  parameters
    NAME = "World"
  end-parameters

  writeln("Hello, ", NAME, "!")
end-model
```

Run with different parameters:

```
import moselpy as mp

model = mp.load_model("greeting.bim")

# Method 1: Dict (recommended)
model.run(exec_params={"NAME": "Python"})

# Method 2: String
model.run(exec_params="NAME=MoselPy")
```

Output:

```
Hello, Python!
Hello, MoselPy!
```

The dict format handles special characters automatically and is the preferred option.

1.4 Stream redirection

Control where Mosel output goes using `set_default_stream()`. See Chapter 6 for the full API reference of `Model.set_default_stream()`.

1.4.1 Silencing output

Redirect output to the built-in `null:` driver to suppress it:

```
import moselpy as mp

model = mp.load_model("model.bim")
model.set_default_stream(mp.StreamType.OUTPUT, "null:")
model.set_default_stream(mp.StreamType.ERROR, "null:")
model.run()
```

1.4.2 Capturing output

Redirect output to a Python buffer using the `moselpy:` driver. Register a `StringIO` object in the model's symbol registry, then route the stream through it. The buffer must be set before calling `run()`:

```
import moselpy as mp
from io import StringIO
model = mp.load_model("model.bim")

buf = StringIO()
model.symbols["out"] = buf
model.set_default_stream(mp.StreamType.OUTPUT, "moselpy:out")
model.run()
del model.symbols["out"]

print(f"Captured: {buf.getvalue()}")
```

Output: `Captured: Hello from Mosel!`

1.4.3 Global vs per-model redirection

Global stream settings affect all models that have not overridden their own streams. Per-model settings take precedence. Note that global settings via `config.set_default_stream()` must be set **before** the first `load_model()` call in the process. Once a model is loaded, only per-model `set_default_stream()` calls are possible:

```
import moselpy as mp

# Global: must be called BEFORE the first load_model()
mp.config.set_default_stream(mp.StreamType.OUTPUT, "null:")

# Per-model: can be called at any time, overrides global for this model only
model = mp.load_model("model.bim")
model.set_default_stream(mp.StreamType.OUTPUT, "output.txt")
```

Pass `None` as the file name to restore the default stream (`sys.stdout`, `sys.stderr`, or `sys.stdin`).

1.5 Running a model from a string

For quick prototyping or generated models, you can define Mosel source directly as a Python string. Since `compile_model()` and `execute_model()` expect a file path, write the source to a temporary

file, compile, run, then clean up:

```
import moselpy as mp
import tempfile, os

src = """model SumToN
  declarations
    n: integer
    result: real
  end-declarations

  initializations from "moselpy:"
    n
  end-initializations

  result := n * (n + 1) / 2.0

  initializations to "moselpy:"
    result
  end-initializations
end-model"""

with tempfile.NamedTemporaryFile(mode="w", suffix=".mos", delete=False) as f:
    f.write(src)
    tmp_mos = f.name
tmp_bim = tmp_mos[:-4] + ".bim"

try:
    mp.compile_model(tmp_mos, tmp_bim)
    model = mp.load_model(tmp_bim)
    output = model.run(input_data={"n": 100})
    print(f"Sum 1..100 = {output['result']:.0f}")
finally:
    for p in [tmp_mos, tmp_bim]:
        if os.path.exists(p):
            os.unlink(p)
```

Output:

```
Sum 1..100 = 5050
```

This pattern is useful for testing small models interactively or for dynamically generating Mosel source from Python.

CHAPTER 2

Data Exchange

2.1 Python to Mosel data flow

MoselPy enables bidirectional data exchange in memory between Python and Mosel through the `moselpy:` I/O driver.

2.1.1 Basic pattern

Data exchange uses Mosel's `initializations` blocks with the `"moselpy:"` driver. In the Mosel model, use `initializations from "moselpy:"` to receive input data from Python and `initializations to "moselpy:"` to send output data back. The following example demonstrates multiple types at once (a string, an integer, and an array):

Note: `initializations from/to "moselpy:"` can be used with local or global entities. For access via `find_identifier()` after execution, the explicit `public` keyword is required; entities are private by default (see Section 3.3). When compiled with the `g` or `G` flag, all globally declared entities become visible to `find_identifier()`.

```
model DataDemo
  uses "mmsystem"

  declarations
    user_name: string
    num_items: integer
    values: array(1..3) of real
    total: real
    message: text
  end-declarations

  initializations from "moselpy:"
    user_name num_items values
  end-initializations

  total := sum(i in 1..3) values(i)
  message := "Hello " + user_name + "! Total is " + total

  initializations to "moselpy:"
    total message
  end-initializations
end-model
```

Pass input as a dictionary to `run()`. The method returns a dictionary of all output values:

```
import moselpy as mp

mp.compile_model("datademo.mos", "datademo.bim")
model = mp.load_model("datademo.bim")
```

```
output = model.run(input_data={
    "user_name": "Alice",
    "num_items": 5,
    "values": {1: 10.0, 2: 20.0, 3: 30.0}
})

print(output['message'])
print(f"Total: {output['total']}")
```

Output:

```
Hello Alice! Total is 60
Total: 60.0
```

See Section 2.4 for how to make this model runnable standalone for testing.

2.2 Type mapping

The sections below show code examples for each type. For the full type conversion table including NumPy and bytes support, see Chapter 8.

2.2.1 Scalars

Scalar types (integer, real, string, boolean) are passed directly as Python values:

```
# All scalar types together
data = {
    "count":  42,        # integer
    "price":  19.99,     # real
    "name":   "Widget",  # string
    "active": True,      # boolean
}
```

2.2.2 Arrays

Mosel arrays are passed as Python dicts, where keys are the array indices and values are the data:

```
# Mosel: values: array(1..3) of real
data = {"values": {1: 10.0, 2: 20.0, 3: 30.0}}

# Built from a list using a loop
values_list = [10.0, 20.0, 30.0]
data = {"values": {i+1: v for i, v in enumerate(values_list)}}
```

For 2D arrays, use tuple keys:

```
# Mosel: array(ROWS, COLS) of real  -- explicit
data = {"matrix": {(1, 1): 5.0, (1, 2): 10.0, (2, 1): 15.0, (2, 2): 20.0}}

# Built from nested lists using loops
rows, cols = [1, 2], [1, 2]
values = [[5.0, 10.0], [15.0, 20.0]]
data = {"matrix": {(r, c): values[i][j]
                   for i, r in enumerate(rows)
                   for j, c in enumerate(cols)}}
```

2.2.3 Sets

Python `set` maps directly to a Mosel `set`. Several common ways to build one:

```
# Mosel: PRODUCTS: set of string
# Literal
data = {"PRODUCTS": {"Widget", "Gadget", "Gizmo"}}

# From a list (preserves no order, as sets are unordered)
products = ["Widget", "Gadget", "Gizmo"]
data = {"PRODUCTS": set(products)}

# From a DataFrame or Series index
data = {"PRODUCTS": set(demand_df.index)}
data = {"MONTHS":   set(capacity_series.index)}

# Integer set from a range
data = {"PERIODS": set(range(1, 13))}  # 1..12
```

2.2.4 Lists

Python `list` maps to a Mosel `list` (an ordered sequence). Pass directly or build from a range:

```
# Literal
data = {"months": ["Jan", "Feb", "Mar", "Apr", "May", "Jun"]}

# From range
data = {"periods": list(range(1, 7))}  # [1, 2, 3, 4, 5, 6]

# From an existing collection
data = {"products": list(demand_df.index)}
```

2.3 Pandas integration

MoselPy converts Pandas Series and DataFrames to Mosel arrays automatically, using the index as the set dimension, for both input and output. In Mosel, declare arrays indexed by a dynamic set; the set is populated automatically when the array data is read:

```
model SupplyDemand
  declarations
    CITIES: set of string
    Demand, Capacity: array(CITIES) of real
    Revenue: array(CITIES) of real   ! output
  end-declarations

  ! CITIES populated automatically from the Series index when reading array data
  initializations from "moselpy:"
    Demand Capacity
  end-initializations

  forall(c in CITIES) Revenue(c) := Demand(c) * 10.0

  initializations to "moselpy:"
    Revenue
  end-initializations
end-model
```

```
import pandas as pd
import moselpy as mp

model = mp.load_model("supply.bim")

# Input: pass Series/DataFrame directly in input_data
```

```
demand   = pd.Series({"NYC": 100, "LA": 80, "CHI": 90})
capacity = pd.Series({"NYC": 120, "LA": 100, "CHI": 85})

output = model.run(input_data={"Demand": demand, "Capacity": capacity})

# Output 1D: wrap dict in pd.Series, or use find_identifier(use_pandas=True)
revenue = pd.Series(output["Revenue"])
# or: revenue = model.find_identifier("Revenue", use_pandas=True)
```

2D arrays. For a 2D Mosel array (*e.g.* `Production: array(PRODUCTS, MONTHS) of real`), `find_identifier(use_pandas=True)` returns a MultiIndex Series. Call `unstack()` to get a DataFrame with rows and columns matching the Mosel index sets:

```
production_series = model.find_identifier("Production", use_pandas=True)
production_df = production_series.unstack(level=1)
print(production_df)
```

Output:

```
             Jan    Feb    Mar    Apr    May    Jun
Gadgets    53.33  53.33  53.33  53.33  53.33  46.67
Gizmos     60.00  70.00  80.00  75.00  65.00  55.00
Widgets     0.00   0.00   0.00   0.00   0.00   0.00
```

2.4 Parameterising the data source

The `initializations` blocks allow switching between data sources of different types by parameterising the source name. Use a string parameter `DATASOURCE` that defaults to a local `.dat` file for standalone execution; when calling from Python, pass `DATASOURCE="moselpy:"` via `exec_params`:

```
parameters
  DATASOURCE = "mydata.dat"   ! default: read from file when running standalone
end-parameters

! ... declarations ...

initializations from DATASOURCE
  x y z
end-initializations

! ... model logic ...

initializations to DATASOURCE
  result
end-initializations
```

Run standalone for testing (reads/writes `mydata.dat`):

```
mosel model.mos
```

Run from Python (uses the `moselpy:` driver for live data exchange):

```
model.run(exec_params={"DATASOURCE": "moselpy:"}, input_data={...})
```

CHAPTER 3

Optimization Workflows

3.1 Running optimization models

We demonstrate with a multi-period production planning problem: a factory produces multiple products over several months, and we want to maximize profit while respecting monthly capacity. This example also shows how Pandas DataFrames and Series are passed as input and how multiple output arrays are returned.

3.1.1 Mosel model

Create `production.mos`. The model receives sets, Series, and a DataFrame from Python through `initializations from "moselpy:"`, and returns four output arrays plus the total profit. Note how Mosel's syntax closely resembles mathematical notation:

```
model "multiperiod_production"
  uses "mmxprs"

  parameters
    DATASOURCE = "production.dat"
  end-parameters

  public declarations
    PRODUCTS: set of string
    MONTHS: set of string

    Profit: array(PRODUCTS) of real
    Demand: array(PRODUCTS, MONTHS) of real
    Capacity: array(MONTHS) of real
    Duration: array(PRODUCTS) of real

    produce: array(PRODUCTS, MONTHS) of mpvar
    CapLimit: array(MONTHS) of linctr

    ! Output arrays
    Production: array(PRODUCTS, MONTHS) of real
    MonthlyRevenue: array(MONTHS) of real
    ProductRevenue: array(PRODUCTS) of real
    CapacityUsed: array(MONTHS) of real
    TotalProfit: real
  end-declarations

  initializations from DATASOURCE
    Profit Demand Capacity Duration
  end-initializations

  ! Upper bound: production cannot exceed demand (lower bound defaults to 0)
  forall(p in PRODUCTS, m in MONTHS) produce(p, m) <= Demand(p, m)

  ! Capacity constraint per month (naming enables getact after solve)
  forall(m in MONTHS)
```

```
    CapLimit(m) := sum(p in PRODUCTS) Duration(p) * produce(p, m) <= Capacity(m)

  ! Maximize profit
  maximize(sum(p in PRODUCTS, m in MONTHS) Profit(p) * produce(p, m))

  ! Store results
  TotalProfit := getobjval
  forall(p in PRODUCTS, m in MONTHS) Production(p, m) := getsol(produce(p, m))
  forall(m in MONTHS) do
    MonthlyRevenue(m) := sum(p in PRODUCTS) Profit(p) * getsol(produce(p, m))
    CapacityUsed(m) := getact(CapLimit(m))
  end-do
  forall(p in PRODUCTS) ProductRevenue(p) := sum(m in MONTHS) Profit(p) * getsol(produce(p, m))

  initializations to DATASOURCE
    TotalProfit Production MonthlyRevenue ProductRevenue CapacityUsed
  end-initializations
end-model
```

3.1.2 Python code

Build input data using Pandas. A DataFrame encodes demand by product and month; Series encode per-product and per-month scalar data. Pass them directly to `run()`. MoselPy converts them automatically, using the index as the set dimension:

```
import pandas as pd
import moselpy as mp

products = ["Widgets", "Gadgets", "Gizmos"]
months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun"]

# Demand varies by product and month (2D -> Mosel array(PRODUCTS, MONTHS))
demand_df = pd.DataFrame({
    "Jan": [100, 80, 60], "Feb": [120, 90, 70], "Mar": [140, 100, 80],
    "Apr": [130, 95, 75], "May": [110, 85, 65], "Jun": [90, 70, 55],
}, index=products)

profit_series   = pd.Series({"Widgets": 25, "Gadgets": 40, "Gizmos": 55})
duration_series = pd.Series({"Widgets": 1.0, "Gadgets": 1.5, "Gizmos": 2.0})
capacity_series = pd.Series({
    "Jan": 200, "Feb": 220, "Mar": 240, "Apr": 230, "May": 210, "Jun": 180
})

mp.compile_model("production.mos", "production.bim")
model = mp.load_model("production.bim")
model.set_default_stream(mp.StreamType.OUTPUT, "null:")

output = model.run(
    exec_params={"DATASOURCE": "moselpy:"},
    input_data={
        "Profit": profit_series,
        "Demand": demand_df,
        "Capacity": capacity_series,
        "Duration": duration_series,
    })

print(f"Total Profit: ${output['TotalProfit']:,.2f}")
```

Output: `Total Profit: $34,808.33`

The full output arrays (`Production`, `MonthlyRevenue`, `ProductRevenue`, `CapacityUsed`) are also available in `output`. See Section 3.3 for how to retrieve and display them.

3.2 Problem status

After running an optimization model, check the execution status using the `exec_status` attribute and the optimization status using the `prob_status` object before trying to access any solution information. The `exec_status` indicates whether the model executed without errors, while `prob_status` indicates whether a solution is available:

```
import moselpy as mp

model = mp.load_model("model.bim")
output = model.run(input_data=data)

# Check execution status
if model.exec_status == mp.ExecStatus.OK:
    print("Model executed successfully")

    # Check optimization status
    if model.prob_status.is_solution_optimal:
        print("Optimal solution found")
    elif model.prob_status.is_problem_infeasible:
        print("Problem is infeasible")
    elif model.prob_status.is_problem_unbounded:
        print("Problem is unbounded")
    else:
        print("No optimal solution found")
else:
    print(f"Execution error: {model.exec_status}")
```

Output:

```
Model executed successfully
Optimal solution found
```

The `prob_status` object provides boolean properties:

- `is_solution_optimal`
- `is_solution_available`
- `is_problem_infeasible`
- `is_problem_unbounded`
- `is_optimization_finished`
- `is_optimization_failed`
- `is_problem_valid`

If the model does not call `maximize` or `minimize`, `prob_status.status_code` returns `-1` and all `is_*` properties are set to `False`. For the full list of status codes, see the Xpress Optimizer Reference Manual.

The objective function value is available directly via `model.objective_value`. The exit code from the Mosel model (set via `exit` statement, defaults to 0 indicating successful execution) is available as `model.exit_code`. See Chapter 6 for the full reference.

3.3 Accessing results

There are two methods to retrieve results after running a model. Always check the `prob_status` object before trying to access any solution information. For converting results to Pandas Series and DataFrames, see Section 2.3.

3.3.1 Method 1: run() return value

The `run()` method returns a dictionary containing all variables explicitly output by the Mosel model via `initializations to "moselpy:"`. Scalars are returned as Python scalars; arrays are returned as dicts.

```
output = model.run(input_data=data)

# Scalar
profit = output['TotalProfit']
print(f"Total Profit: ${profit:,.2f}")

# 1D array -> dict
product_revenue = output['ProductRevenue']  # {'Gadgets': ..., 'Gizmos': ..., 'Widgets': ...}

# 2D array -> dict with tuple keys
production = output['Production']  # {('Gadgets','Jan'): ..., ('Gizmos','Jan'): ..., ...}
```

Output:

```
Total Profit: $34,808.33
```

Only variables listed in the Mosel `initializations to "moselpy:"` block are included in the output dict. Use Method 2 to access any other model entity after the solve.

3.3.2 Method 2: find_identifier()

`find_identifier()` retrieves the value of any public model entity by name after a solve, including variables not in the output block. It can return raw Python values or Pandas objects. Entities must be declared with the explicit `public` keyword in the Mosel source. When the model is compiled with the `g` or `G` flag, all global symbols become visible:

```
model.run(input_data=data)

# Scalar
profit = model.find_identifier("TotalProfit")
print(f"TotalProfit: {profit:.2f}")

# 1D array as dict (default)
product_revenue = model.find_identifier("ProductRevenue")
print(product_revenue)

# 1D array as Pandas Series
product_revenue_series = model.find_identifier("ProductRevenue", use_pandas=True)
print(product_revenue_series)
```

Output:

```
TotalProfit: 34808.33
{'Gadgets': 12533.33, 'Gizmos': 22275.0, 'Widgets': 0.0}
Gadgets    12533.33
Gizmos     22275.00
Widgets        0.00
dtype: float64
```

Use `find_identifier()` when you need to inspect intermediate variables or parameters that are not part of the model output, for example for debugging or post-solve analysis.

3.4 Visualizing results

Optimization results from MoselPy integrate naturally with matplotlib, enabling production dashboards and analysis charts. This section continues the multi-period production example from the previous section.

3.4.1 Retrieving results as Pandas objects

Use `find_identifier()` with the `use_pandas=True` flag to retrieve output arrays as Pandas objects. For 1D arrays it returns a Series; for 2D arrays it returns a Series with a MultiIndex, which can be unstacked into a DataFrame:

```
# 1D arrays -> Series
monthly_revenue = model.find_identifier("MonthlyRevenue", use_pandas=True)
product_revenue = model.find_identifier("ProductRevenue", use_pandas=True)
capacity_used   = model.find_identifier("CapacityUsed", use_pandas=True)

# 2D array -> MultiIndex Series, unstack to DataFrame
production_df = model.find_identifier("Production", use_pandas=True).unstack(level=1)
print(production_df)
```

Output:

```
            Jan    Feb    Mar    Apr    May    Jun
Widgets    0.00   0.00   0.00   0.00   0.00   0.00
Gadgets   53.33  53.33  53.33  53.33  53.33  46.67
Gizmos    60.00  70.00  80.00  75.00  65.00  55.00
```

3.4.2 Production dashboard

Plot a four-panel chart: monthly production by product (stacked bars), revenue share by product (pie), capacity utilization over time (fill chart), and monthly revenue trend (bar chart):

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(2, 2, figsize=(12, 10))

# 1. Production by product (stacked bar)
production_df.T.plot(kind="bar", stacked=True, ax=axes[0, 0], colormap="viridis")
axes[0, 0].set_title("Monthly Production by Product")
axes[0, 0].set_xlabel("Month")
axes[0, 0].set_ylabel("Units")
axes[0, 0].tick_params(axis='x', rotation=45)

# 2. Revenue by product (pie)
product_revenue_ordered = product_revenue.reindex(products)
axes[0, 1].pie(product_revenue_ordered, labels=products,
               autopct='%1.1f%%', startangle=90)
axes[0, 1].set_title("Revenue Share by Product")

# 3. Capacity utilization (fill + line)
capacity_ordered = capacity_series.reindex(months)
used_ordered = capacity_used.reindex(months)
x = range(len(months))
axes[1, 0].fill_between(x, capacity_ordered, alpha=0.3, label="Available")
axes[1, 0].plot(x, used_ordered, 'o-', linewidth=2, markersize=8, label="Used")
axes[1, 0].set_xticks(x)
axes[1, 0].set_xticklabels(months)
axes[1, 0].set_title("Capacity Utilization")
axes[1, 0].legend()

# 4. Monthly revenue (bar)
```

```
monthly_ordered = monthly_revenue.reindex(months)
axes[1, 1].bar(months, monthly_ordered, color="steelblue")
axes[1, 1].set_title("Monthly Revenue")
axes[1, 1].set_ylabel("Revenue ($)")

plt.tight_layout()
plt.show()
```

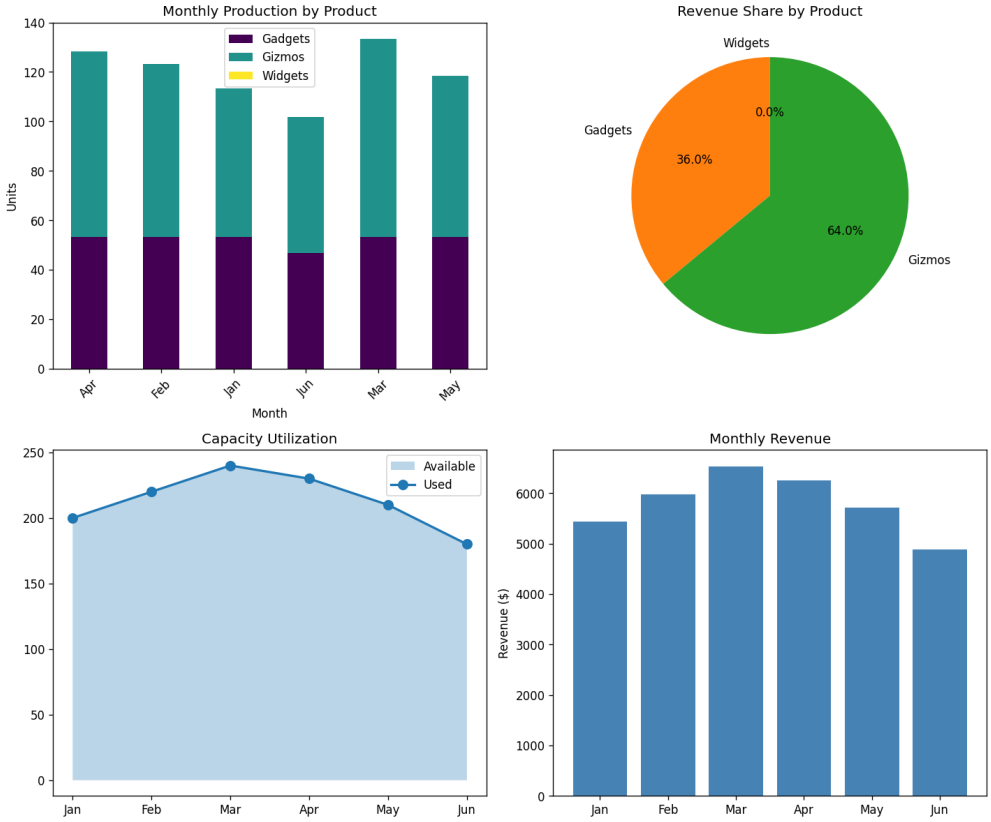


Figure 3.1: Production dashboard: monthly production by product, revenue share, capacity utilization, and monthly revenue

Chapter 4

Advanced Topics

4.1 Iterative solving

It is possible to run the same compiled model multiple times with different inputs, for example to perform sensitivity analysis and scenario comparison. The package allows you to load a model once with `load_model()` and call `run()` repeatedly, each time supplying a different `input_data` dictionary. Each call solves a fresh instance of the problem and returns an independent output dictionary. Results across scenarios can be collected and analysed using standard Python data structures or Pandas.

4.1.1 Capacity scenario analysis

The following example tests four capacity levels (80%, 100%, 120%, and 150% of baseline) using the multi-period production model. Only the `Capacity` input varies between scenarios; all other inputs remain constant. The model is loaded once; `run()` automatically resets state between calls. Loading a model inside a loop is not recommended as it re-parses the BIM file on each iteration:

```
import moselpy as mp
import pandas as pd
import matplotlib.pyplot as plt

capacity_multipliers = [0.8, 1.0, 1.2, 1.5]
scenario_profits = {}

model = mp.load_model("production.bim")
model.set_default_stream(mp.StreamType.OUTPUT, "null:")
model.set_default_stream(mp.StreamType.ERROR, "null:")

for mult in capacity_multipliers:
    output = model.run(
        exec_params={"DATASOURCE": "moselpy:"},
        input_data={
            "Profit": profit_series,
            "Demand": demand_df,
            "Capacity": capacity_series * mult,
            "Duration": duration_series,
        })

    scenario_profits[f"{mult:.0%}"] = output["TotalProfit"]

# Build a Series for easy analysis
results = pd.Series(scenario_profits, name="Total Profit")
print(results.apply(lambda x: f"${x:,.0f}"))
```

Output:

```
80%     $27,982
100%    $34,808
120%    $41,633
```

```
150%    $51,325
```

4.1.2 Plotting scenario results

Visualize how profit scales with capacity. The `pd.Series` built above provides a ready-made source for a bar chart:

```
fig, ax = plt.subplots(figsize=(8, 5))
bars = ax.bar(results.index, results.values, edgecolor="black")

for bar, val in zip(bars, results.values):
    ax.text(bar.get_x() + bar.get_width() / 2, bar.get_height() + 500,
            f"${val:,.0f}", ha="center", va="bottom", fontweight="bold")

ax.set_title("Profit vs Capacity Level")
ax.set_xlabel("Capacity (% of baseline)")
ax.set_ylabel("Total Profit ($)")
ax.set_ylim(0, max(results.values) * 1.15)
plt.tight_layout()
plt.show()
```

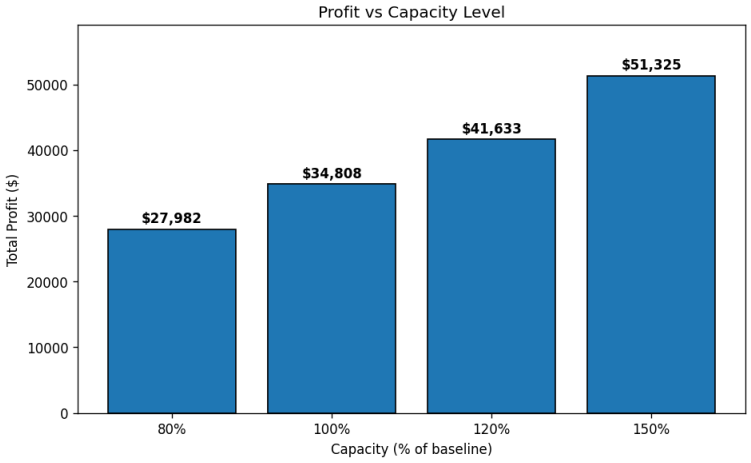


Figure 4.1: Profit vs capacity level for four capacity scenarios

4.2 Model control

The package provides methods to control running models from Python. This is useful for implementing timeouts, user-initiated cancellation, or coordinating multiple model runs.

4.2.1 Graceful stop

Use `model.stop()` to request the model to stop at the next control flow point (loop, conditional, procedure call). The Mosel runtime checks this flag automatically:

```
import moselpy as mp

model = mp.load_model("long_running.bim")

# From a callback or timer thread
if model.is_running:
    stopped = model.stop()  # Returns True if stop was requested
```

```
    print(f"Stop requested: {stopped}")

# After execution
if model.exec_status == mp.ExecStatus.STOP:
    print("Model was stopped by user")
```

4.2.2 Immediate termination

Use `model.terminate()` for immediate termination without waiting for a safe stopping point. Use with caution as it may leave model state inconsistent:

```
model.terminate()  # Immediate termination
```

4.3 Model introspection

The MoselPy package allows you to inspect compiled model metadata directly from Python without running the model. Properties such as `name`, `version`, `size`, and `user_comment` are available on any loaded model object. It is also possible to retrieve custom metadata embedded in the Mosel source via `get_annotations()`, and to query encryption and signing status through the `sec_status` object. This is useful for validation, logging, and build pipelines that need to verify model provenance:

```
import moselpy as mp

model = mp.load_model("model.bim")

print(f"Model name: {model.name}")
print(f"BIM version: {model.version}")
print(f"File size: {model.size} bytes")
print(f"User comment: {model.user_comment}")

# Security status
if model.sec_status.is_encrypted:
    print("Model is encrypted")
if model.sec_status.is_signed:
    print(f"Signed with key: {model.key_fingerprint}")
```

Output:

```
Model name:   multiperiod_production
BIM version:  0
File size:    18888 bytes
User comment:
```

4.4 Annotations

Annotations are custom metadata defined in Mosel source using `!@` directives. Any key-value pair can be defined and will be available via `get_annotations()` at runtime. Annotations are organized in dot-separated categories (*e.g.*, `doc.descr`, `insight.alias`).

```
model "annotated"
!@description: Production planning model
!@author: Team Name
!@version: 1.2
!@mc.flush                        ! Not associated with declarations block
  declarations
    !@doc.descr The integer variable
    public n: integer
  end-declarations
  n := 42
end-model
```

Read annotations from Python (compile with the `D` flag to record `doc`-category annotations):

```
mp.compile_model("model.mos", "model.bim", "D")
model = mp.load_model("model.bim")

# Model-level annotations
ann = model.get_annotations()
for key, value in ann.items():
    print(f"  {key}: {value}")

# Annotations for a specific public identifier
n_ann = model.get_annotations(ident="n")
print(n_ann.get("doc.descr"))    # "The integer variable"

# Filter by prefix
doc_ann = model.get_annotations(prefix="doc")
```

Output:

```
  .author: Team Name
  .description: Production planning model
  .version: 1.2
  doc.name: annotated
  doc.version: 0.0.0
The decision variable
```

Use `!@mc.flush` to indicate that the preceding annotations are not meant to be associated with the following block (declarations, subroutine definitions, etc.).

4.5 Context managers

The package allows you to use Python's `with` statement to manage model resources automatically. When the `with` block exits (whether normally or due to an exception), the model is released and any associated resources are cleaned up. This is the recommended pattern for short-lived model executions where you do not need to retain the model object afterwards:

```
import moselpy as mp

with mp.load_model("model.bim") as model:
    output = model.run(input_data=data)
    print(output['result'])
# Model resources automatically released on exit
```

4.6 Performance considerations

4.6.1 Binary format for large data

It is possible to significantly reduce data transfer overhead by using binary format for large datasets. The package allows you to serialize Python data to a compact binary representation before passing it to Mosel. For datasets with thousands of elements, this can be measurably faster than the default text-based exchange.

For **file-based** transfer, use `write_binary()` to serialize to a file and `read_binary()` to deserialize. The binary file can be loaded directly in Mosel using the `bin:` driver:

```
import moselpy as mp

data = {"values": {1: 10, 2: 20, 3: 30}, "multiplier": 2.5}
mp.write_binary("data.bin", data)
```

```
# In Mosel:
# initializations from "bin:data.bin"
#   values multiplier
# end-initializations

loaded = mp.read_binary("data.bin")
```

For **in-memory** transfer without files, use `to_binary()` to serialize data to bytes and pass the bytes object directly in `input_data`. Use `from_binary()` to deserialize output bytes:

```
import moselpy as mp

large_data = {"values": {i: random.random() for i in range(100000)}}
bin_data = mp.to_binary(large_data)
output = model.run(input_data={"data": bin_data})
result_data = mp.from_binary(output["output_bytes"])
```

Binary format is compatible with Mosel's `bin:` I/O driver; use `"bin:moselpy:mydata"` for accessing the data from Mosel. See Chapter 8 for more on the I/O driver and Chapter 11 for the full binary format API.

4.7 Exporting problems

The model's optimization problem can be exported to LP or MPS format for debugging, external analysis, or solver comparison:

```
model = mp.load_model("model.bim")
model.run(input_data=data)

# Export to LP format (default)
model.export_problem(file_name="problem.lp")

# Export to MPS format with maximization
model.export_problem(options="mp", file_name="problem.mps")

# Export to console (no file_name)
model.export_problem()
```

See Chapter 6 for the full `export_problem()` reference including all format options.

II. API Reference

CHAPTER 5

Configuration

The package provides a global `config` object of type `Config` for configuring the Mosel environment, and a set of read-only package-level attributes.

Config

Purpose

Configuration class for the Mosel environment. A singleton instance is available as `mp.config`.

Synopsis

```
class Config
```

Further information

1. The `Config` class provides properties and methods for configuring global Mosel settings, including stream redirection, working directories, and other environment parameters.
2. A singleton instance is created when the package is imported and is accessible as `mp.config`. The constructor is available but typically not needed.

Module

moselpy

5.1 Config properties

The following properties are accessible on the `config` object (*e.g.* `mp.config.bim_prefix`).

`bim_prefix`	List of prefixes for locating packages when compiling or loading.	p. 27
`default_working_directory`	Default current working directory for Mosel operations.	p. 27
`locale_directory`	Directory containing translated messages (native language support).	p. 27
`max_stack_dump_depth`	Maximum depth of stack dump in error messages.	p. 27

bim_prefix

Description List of prefixes for locating packages when compiling or loading.

Type String, read/write

Default value `""` (value of environment variable `MOSEL_BIM` if set)

Note Used to locate packages when compiling or loading. The file name is constructed by using each element of the list as a prefix, separated by `||`. For instance, if `bim_prefix` is `"tmp:||./"` and package `mypkg` is needed, the loader tries `tmp:mypkg.bim`, then `./mypkg.bim`.

default_working_directory

Description Default current working directory for Mosel operations.

Type String, read/write

Default value `""` (process current working directory)

Notes

1. Path resolution for non-absolute file names is relative to this directory. Also used as the initial working directory when starting model execution.
2. If set to `None` or empty string, uses the process's current working directory as provided by the operating system.

locale_directory

Description Directory containing translated messages (native language support).

Type String, read/write

Note Used for Mosel's Native Language Support (NLS) system. If your models use translated messages (via Mosel's `gettext` mechanism), this specifies where the compiled translation files (`.mo` files) are located. Most applications do not need to modify this setting.

max_stack_dump_depth

Description Maximum depth of stack dump in error messages.

Type Integer, read/write

Default value 0 (stack dumps disabled)

Notes

1. After a runtime error, Mosel may report the content of the call stack for debugging. This controls how many levels are displayed.
2. Using a value smaller than 1 disables stack dumps.

Config.set_default_stream

Purpose

Set the package-level default stream for all models. For per-model overrides, use `Model.set_default_stream`.

Synopsis

```
procedure config.set_default_stream(wdm, file_name=None)
```

Arguments

`wdm`	Stream type (`StreamType` value: `OUTPUT`, `ERROR`, `INPUT`)
`file_name`	Driver path for the stream. Use `"null:"` to silence, or `"moselpy:name"` to route to a Python object registered in `mp.symbols`. Pass `None` to restore the system default (`sys.stdout`, `sys.stderr`, or `sys.stdin`).

Further information

1. Sets the global default for all models that have not overridden their own streams via `Model.set_default_stream`.
2. This method must be called **before** the first `load_model` call in the process. Once a model has been loaded, subsequent calls to `config.set_default_stream()` will fail. To redirect streams after models are loaded, use `Model.set_default_stream` on each model instance instead.

Related topics

`StreamType`, `mp.symbols`, `Model.set_default_stream`

Module

moselpy

5.2 Package attributes

The following attributes are available at the package level (*e.g.* `mp.version`). All are read-only except `symbols`.

`bim_version`	BIM file format version number.	p. 30
`config`	Global `Config` instance for environment configuration.	p. 30
`library_path`	Path to the Mosel library directory.	p. 30
`mosel_version`	Mosel version as an integer (*e.g.* 6012000 for version 6.12.0).	p. 30
`native_version`	Native interface version number.	p. 30
`symbols`	Global symbol registry used by the `moselpy:` I/O driver.	p. 31
`version`	Version string of the Mosel runtime (*e.g.* `"6.12.0"`).	p. 31
`xpress_version`	Xpress suite version as an integer.	p. 31

bim_version

Description BIM file format version number.

Type Integer, read only

config

Description Global `Config` instance for environment configuration., read only

library_path

Description Path to the Mosel library directory.

Type String, read only

mosel_version

Description Mosel version as an integer (*e.g.* 6012000 for version 6.12.0).

Type Integer, read only

native_version

Description Native interface version number.

Type Integer, read only

symbols

Description Global symbol registry used by the `moselpy:` I/O driver., read/write

Notes

1. Register Python objects by name; reference them as `"moselpy:name"` in stream and file paths. Pre-registered: `sys` (so `"moselpy:sys.stdout"` resolves lazily to `sys.stdout`).
2. See `Model.symbols` for per-model registries that take precedence over the global registry.

version

Description Version string of the Mosel runtime (*e.g.* `"6.12.0"`).

Type String, read only

xpress_version

Description Xpress suite version as an integer.

Type Integer, read only

Chapter 6

Model Class

The `Model` class represents a loaded Mosel model. Model objects are created by `load_model` or `execute_model` and provide methods for running the model and accessing its data.

`Model`	Represents a loaded Mosel model. Provides methods for execution, data access, and stream management.	p. 33
`Model.exec_status`	Execution status from the last `Model.run` call.	p. 45
`Model.exit_code`	Exit code from the last model execution.	p. 46
`Model.export_problem`	Export the active optimization problem to a file.	p. 39
`Model.find_identifier`	Retrieve the value of a public Mosel entity from the model's dictionary.	p. 35
`Model.get_annotations`	Retrieve model annotations (metadata). Annotations can be accessed after loading a model, without running it.	p. 40
`Model.is_loaded`	Whether the model is currently loaded.	p. 43
`Model.is_running`	Whether the model is currently executing.	p. 44
`Model.key_fingerprint`	Fingerprint of the signing key if the model is signed.	p. 50
`Model.name`	The model name as declared in the Mosel source.	p. 41
`Model.number`	Unique numeric identifier for this model instance.	p. 42
`Model.objective_value`	Objective function value from the last optimization.	p. 47
`Model.prob_status`	Problem status object with optimization result information.	p. 48
`Model.reset`	Reset the model to its initial state.	p. 38
`Model.run`	Execute the model with optional input data.	p. 34
`Model.sec_status`	Security status object with model signing/encryption information.	p. 49
`Model.set_default_stream`	Set a default stream for this model.	p. 56
`Model.size`	Size of the loaded model in bytes.	p. 51
`Model.stop`	Request the running model to stop execution gracefully.	p. 36
`Model.symbols`	Per-model symbol registry for the `moselpy:` I/O driver.	p. 53
`Model.sys_comment`	System comment embedded in the BIM file during compilation.	p. 54
`Model.terminate`	Forcefully terminate model execution.	p. 37
`Model.user_comment`	User comment embedded in the BIM file during compilation.	p. 55
`Model.version`	Version number of the loaded model.	p. 52

Model

Purpose

Represents a loaded Mosel model. Provides methods for execution, data access, and stream management.

Synopsis

```
class Model
```

Further information

1. Model objects are created by `load_model` or `execute_model`. They cannot be instantiated directly.
2. Model objects support the context manager protocol and can be used with `with` statements for automatic cleanup.
3. Model objects can be copied using `copy.copy()` or `copy.deepcopy()`. Copying creates an independent clone of the model's execution state using Mosel's built-in cloning mechanism. The clone can be run independently without affecting the original. Note that stream redirections are **not** copied; the clone starts with default stream settings.

Example

Using Model as a context manager:

```
with mp.load_model("model.bim") as model:
    model.run()
    print(model.objective_value)
# Model is automatically cleaned up here
```

Related topics

`load_model`, `execute_model`

Module

moselpy

Model.run

Purpose

Execute the model with optional input data.

Synopsis

```
function model.run(input_data=None, exec_params=None) -> dict
```

Arguments

`input_data`	Dictionary of input data to pass to the model via the `moselpy:` driver. Keys are identifier names, values are Python objects convertible to Mosel types.
`exec_params`	Execution parameters as a string (`"PARAM1=val1,PARAM2=val2"`) or list of strings (`["PARAM1=val1", "PARAM2=val2"]`). Sets values for model parameters (defined in the `parameters` block) and control parameters of modules/packages (*e.g.*, `"mypkg.param=value"`). The special parameter `workdir` defines the initial working directory.

Return value

Dictionary of output data collected via `initializations to "moselpy:"`

Example

Run with input and output data:

```
model = mp.load_model("model.bim")
output = model.run(
    input_data={"x": 10, "arr": {1: 100, 2: 200}},
    exec_params={"VERBOSE": 1, "workdir": "/tmp"}
)
print(output["result"])
```

Further information

1. Input data is made available to the Mosel model via the `moselpy:` I/O driver in `initializations from "moselpy:"` blocks. No special compile options are required: the driver works with normally compiled models (no debug flags needed).
2. Output data is collected from `initializations to "moselpy:"` blocks and returned as a dictionary.
3. After execution, check `Model.exec_status` for execution status and `Model.exit_code` for the result value (from Mosel's `exit` statement, or 0 if not called). The exit_code is only meaningful when exec_status is `ExecStatus.OK`.
4. A model can be run multiple times. `Model.reset` is automatically called before each run to release resources from the previous execution.
5. **Threading:** The module holds Python's Global Interpreter Lock (GIL) during execution. Multiple Model instances can exist, but only one can execute at a time within the same process. For parallel model execution, use Mosel's `mmjobs` module within a single model or run separate Python processes.

Related topics

`Model.exec_status`, `Model.exit_code`, `Model.reset`

Module

moselpy

Model.find_identifier

Purpose

Retrieve the value of a public Mosel entity from the model's dictionary.

Synopsis

```
function model.find_identifier(identifier, use_pandas=None) -> object
```

Arguments

`identifier` Name of the Mosel entity (variable, array, set, etc.)

`use_pandas` If `True`, return arrays as pandas Series/DataFrame. If `False` or `None` (default), return as Python dict.

Return value

The value of the entity, converted to an appropriate Python type

Example

Read entities after execution:

```
model.run()
n = model.find_identifier("n")              # Scalar (integer)
result = model.find_identifier("result")     # Array as dict
series = model.find_identifier("result", True) # Array as pandas Series
```

Further information

1. The entity must be declared with the explicit `public` keyword in the Mosel source (either via a `public declarations` block or an individual `public` qualifier). A `RuntimeError` is raised if the entity is not found.
2. Can retrieve scalars (`integer`, `real`, `boolean`, `string`, `text`), arrays, sets, and lists. When the model is compiled with debug information (flags `-g` or `-G`), both public and private symbols can be accessed.
3. Type conversion follows standard Mosel-to-Python mappings: `integer` to `int`, `real` to `float`, `string` to `str`, `boolean` to `bool`, `mpvar` and `linctr` to their solution values, arrays to `dict` or pandas Series, sets to Python sets. The `text` type from `mmsystem` is returned as its string representation. Other external types from Mosel modules (including `date` and `time` from `mmsystem`) are not supported and raise a `RuntimeError`.
4. Procedures and functions in the model's dictionary cannot be retrieved as Python objects. Union types are also not supported. Attempting to retrieve these raises a `RuntimeError`.

Related topics

`Model.run`, `Model.get_annotations`

Module

moselpy

Model.stop

Purpose

Request the running model to stop execution gracefully.

Synopsis

```
function model.stop() -> bool
```

Return value

`True` if the model was running and stop was requested, `False` if the model was not running

Further information

1. This sends a stop signal to the model. The Mosel runtime checks this flag automatically at control flow points (loops, conditionals, procedure calls). The model will stop at the next such point without requiring explicit checks in Mosel code.

2. For immediate termination without waiting for a control flow point, use `Model.terminate` instead.

Related topics

`Model.terminate`, `Model.is_running`

Module

moselpy

Model.terminate

Purpose

Forcefully terminate model execution.

Synopsis

```
procedure model.terminate()
```

Further information

1. Unlike `Model.stop`, this immediately terminates execution without waiting for the model to reach a safe stopping point.
2. Use with caution as it may leave model state inconsistent.

Related topics

`Model.stop`, `Model.is_running`

Module

moselpy

Model.reset

Purpose

Reset the model to its initial state.

Synopsis

```
procedure model.reset()
```

Further information

1. Releases all resources allocated by the model during execution. The model returns to its state just after loading.
2. This function is automatically called before a model is unloaded or run.

Related topics

`Model.run`

Module

moselpy

Model.export_problem

Purpose

Export the active optimization problem to a file.

Synopsis

```
procedure model.export_problem(options=None, file_name=None)
```

Arguments

`options`	Output format options (can be combined): LP format, minimization (default) `m` MPS format `p` Maximization (default is minimization) `s` Use scrambled names `x` Hexadecimal numbers for MPS
`file_name`	Output file path. If `None`, output goes to console. If no extension given, `.mps` or `.lp` is added based on format.

Further information

1. The model must have a valid optimization problem loaded. This requires running the model first to build the problem.
2. Raises `RuntimeError` if no problem is available for export or in case of I/O error.
3. Exports core Mosel problem objects (equivalent to Mosel's `exportprob`). Extended problem information from solver modules (such as `mmxprs` or `mmnl`/`mmxnlp`) is not included. MPS format does not preserve lower bounds on semi-continuous variables; use LP format if this information is needed.

Module

moselpy

Model.get_annotations

Purpose

Retrieve model annotations (metadata). Annotations can be accessed after loading a model, without running it.

Synopsis

```
function model.get_annotations(ident=None, prefix=None) -> dict
```

Arguments

`ident`	Filter by identifier name (optional)
`prefix`	Filter by annotation prefix (optional)

Return value

Dictionary mapping annotation names to values

Further information

1. Annotations are metadata defined in the Mosel source using special comments: `!@name value` for single-line annotations. They are organized in dot-separated categories (*e.g.*, `doc.descr`, `insight.alias`).

2. Use `!@mc.flush` to mark the end of model-level annotations before any immediately following block (declarations, subroutine definitions, etc.).

3. Annotations can be global (model-level) or associated with specific public symbols depending on their location in the source. The `-D` compiler option records annotations of the `doc` category; annotations in other categories (*e.g.*, `insight`) are stored independently of this flag.

Module

moselpy

Model.name

Purpose

The model name as declared in the Mosel source.

Synopsis

```
model.name -> str
```

Return value

Model name string (read-only property)

Module

moselpy

Model.number

Purpose

Unique numeric identifier for this model instance.

Synopsis

```
model.number -> int
```

Return value

Model number (read-only property)

Module

moselpy

Model.is_loaded

Purpose

Whether the model is currently loaded.

Synopsis

```
model.is_loaded -> bool
```

Return value

`True` if the model is loaded, `False` otherwise

Further information

Becomes `False` after the model is unloaded (*e.g.*, when exiting a context manager or when explicitly destroyed).

Module

moselpy

Model.is_running

Purpose

Whether the model is currently executing.

Synopsis

```
model.is_running -> bool
```

Return value

`True` if execution is in progress, `False` otherwise

Related topics

`Model.stop`, `Model.terminate`

Module

moselpy

Model.exec_status

Purpose

Execution status from the last `Model.run` call.

Synopsis

```
model.exec_status -> ExecStatus
```

Return value

An `ExecStatus` enumeration value

Further information

Common values: `ExecStatus.OK` (successful), `ExecStatus.ERROR` (runtime error), `ExecStatus.STOP` (stopped by user).

Related topics

`ExecStatus`, `Model.exit_code`

Module

moselpy

Model.exit_code

Purpose

Exit code from the last model execution.

Synopsis

```
model.exit_code -> int
```

Return value

Integer exit code (0 indicates successful execution)

Further information

The exit code is set by the Mosel model via the `exit` statement or defaults to 0 for normal completion.

Related topics

`Model.exec_status`

Module

moselpy

Model.objective_value

Purpose

Objective function value from the last optimization.

Synopsis

```
model.objective_value -> float
```

Return value

Objective value as a floating-point number

Further information

Returns 0.0 if no optimization has been performed or if the problem has no solution.

Related topics

`Model.prob_status`

Module

moselpy

Model.prob_status

Purpose

Problem status object with optimization result information.

Synopsis

```
model.prob_status -> ProblemStatus
```

Return value

A `ProblemStatus` object

Further information

Alias: `model.problem_status` returns the same object.

Related topics

`ProblemStatus`, `Model.objective_value`

Module

moselpy

Model.sec_status

Purpose

Security status object with model signing/encryption information.

Synopsis

```
model.sec_status -> SecurityStatus
```

Return value

A `SecurityStatus` object

Related topics

`SecurityStatus`, `Model.key_fingerprint`

Module

moselpy

Model.key_fingerprint

Purpose

Fingerprint of the signing key if the model is signed.

Synopsis

```
model.key_fingerprint -> str | None
```

Return value

Key fingerprint string, or `None` if not signed

Related topics

`Model.sec_status`

Module

moselpy

Model.size

Purpose

Size of the loaded model in bytes.

Synopsis

```
model.size -> int
```

Return value

Model size in bytes

Module

moselpy

Model.version

Purpose

Version number of the loaded model.

Synopsis

```
model.version -> int
```

Return value

Version number

Further information

The version is specified via the `version` option in the Mosel source (*e.g.* `version 1.2.3`) as a plain integer (*e.g.* 1200300). This is the programmer-defined model version and is distinct from both the BIM file format version and the moselpy library version (see `mp.version`).

Module

moselpy

Model.symbols

Purpose

Per-model symbol registry for the `moselpy:` I/O driver.

Synopsis

```
model.symbols -> dict
```

Return value

A dict mapping names to Python objects for this model's streams

Further information

Objects registered here are accessible via `"moselpy:name"` paths scoped to this model only. Use this to isolate stream targets when running multiple models concurrently. For global registration, use `mp.symbols`.

Related topics

`Model.set_default_stream`, `mp.symbols`

Module

moselpy

Model.sys_comment

Purpose

System comment embedded in the BIM file during compilation.

Synopsis

```
model.sys_comment -> str
```

Return value

System comment string (contains compilation info such as compiler version and date)

Related topics

`Model.user_comment`

Module

moselpy

Model.user_comment

Purpose

User comment embedded in the BIM file during compilation.

Synopsis

```
model.user_comment -> str
```

Return value

User comment string

Further information

Set via the `user_comment` parameter of `compile_model`.

Related topics

`compile_model`, `Model.sys_comment`

Module

moselpy

Model.set_default_stream

Purpose

Set a default stream for this model.

Synopsis

```
procedure model.set_default_stream(wdm, file_name=None)
```

Arguments

`wdm`	Stream type (`StreamType` value: `OUTPUT`, `ERROR`, `INPUT`)
`file_name`	Driver path for the stream. Use `"null:"` to silence, or `"moselpy:name"` to route to a Python object registered in `Model.symbols`. Pass `None` to restore the global default.

Example

Silence output, capture to buffer, redirect to file:

```
# Silence
model.set_default_stream(mp.StreamType.OUTPUT, "null:")

# Capture to a StringIO buffer
from io import StringIO
buf = StringIO()
model.symbols["out"] = buf
model.set_default_stream(mp.StreamType.OUTPUT, "moselpy:out")
model.run()
del model.symbols["out"]
print(buf.getvalue())

# Redirect to a file
model.set_default_stream(mp.StreamType.OUTPUT, "output.txt")
```

Further information

1. Must be called before `Model.run` to take effect. Overrides the global stream set via `Config.set_default_stream` for this model only.
2. Both `OUTPUT` and `ERROR` streams can be routed to the same Python object. Note that they flush independently and output may interleave.

Related topics

`StreamType`, `Model.symbols`, `Config.set_default_stream`

Module

moselpy

CHAPTER 7

Package Functions

`compile_model`	Compile a Mosel source file to a binary model (BIM) file.	p. 58
`execute_model`	Compile and execute a Mosel source file in one step.	p. 60
`load_model`	Load a compiled binary model (BIM) file.	p. 59
`remove_temp_directory`	Remove the Mosel temporary directory.	p. 61

compile_model

Purpose

Compile a Mosel source file to a binary model (BIM) file.

Synopsis

```
procedure compile_model(src_file, dst_file=None, options=None,
      user_comment=None, password_file=None, private_key=None,
      public_keys_file=None)
```

Arguments

`src_file` — Path to the Mosel source file (`.mos`). May contain environment variable references using `${varname}` notation.

`dst_file` — Path for the output BIM file. If `None`, uses the source file path with `.bim` extension.

`options` — Compiler options string (options separated by spaces or – symbols):

`g`	Include debugging information (error locations, all symbols visible to `Model.find_identifier`)
`G`	Include tracing information (step-by-step debugging, all symbols visible to `Model.find_identifier`)
`p`	Parse only (syntax analysis, no output file)
`ni`	Disable implicit declarations
`na`	Disable assertions in debug mode
`wi`	Warn on implicit declarations
`wu`	Report unused public symbols
`we`	Treat warnings as errors
`D`	Generate and record documentation annotations
`S`	Sign the BIM file
`E`	Encrypt the BIM file
`F`	Interpret password_file as file name (not password itself)
`V`	Only load signed packages with verified signatures
`T`	Only load signed packages with valid signatures

`user_comment` — Commentary text saved at the beginning of the BIM file

`password_file` — Password or password file for encryption

`private_key` — Private key file for BIM file signing

`public_keys_file` — File of public keys (each line is a key file name) for encryption with multiple recipients

Example

Compile with documentation annotations:

```
mp.compile_model("model.mos", "model.bim", "D")
```

Further information

1. Raises an exception if compilation fails with one of the following causes: syntax error or file access error, semantic error, error writing output file, or license error.
2. When using public_keys_file for encryption, the BIM file is encrypted for each listed public key, allowing decryption by any corresponding private key.

Related topics

`load_model`, `execute_model`

Module

moselpy

load_model

Purpose

Load a compiled binary model (BIM) file.

Synopsis

```
function load_model(bim_path, flags=None, password_file=None,
      private_key_file=None, public_keys_file=None) -> Model
```

Arguments

`bim_path`	Path to the BIM file to load
`flags`	Loading options string: `c` Check signature (if the file is signed) `V` Load signed files only if signature is valid `T` Load only signed files with valid signature `F` Interpret password_file as file name (not password itself) `l` Do not load required packages `h` Load only header of the BIM file
`password_file`	Password or password file for encrypted BIM files
`private_key_file`	Private key file for encrypted BIM files
`public_keys_file`	File of public keys for signature verification

Return value

A `Model` object ready for execution

Example

Load and run a model:

```
model = mp.load_model("model.bim")
model.run()
```

Further information

1. Creates a new model instance from the BIM file. While loading, Mosel automatically loads any Mosel modules (DSO) and packages required by the model.
2. Both models and packages can be loaded, but only models can be executed with `Model.run`.
3. The returned Model object must be kept alive while the model is in use. When the Model object is destroyed, the underlying Mosel model is unloaded.

Related topics

`compile_model`, `execute_model`, `Model`

Module

moselpy

execute_model

Purpose

Compile and execute a Mosel source file in one step.

Synopsis

```
function execute_model(src_file, options=None, exec_params=None,
       input_data=None) -> tuple[Model, dict]
```

Arguments

`src_file`	Path to the Mosel source file (`.mos`)
`options`	Compiler options string
`exec_params`	Execution parameters as a dictionary of `str` keys and values
`input_data`	Input data dictionary for the `moselpy:` I/O driver

Return value

A tuple `(model, output)` where `model` is the `Model` object and `output` is a dictionary of data written by the model via `initializations to "moselpy:"`

Example

Quick execution:

```
model, output = mp.execute_model("model.mos")
print(model.exec_status)
```

Further information

1. This is a convenience function that combines compilation and execution. The model is compiled to an in-memory BIM, loaded, and executed.
2. For production use or repeated execution, prefer `compile_model` followed by `load_model` to avoid recompilation overhead.

Related topics

`compile_model`, `load_model`, `Model.run`

Module

moselpy

remove_temp_directory

Purpose

Remove the Mosel temporary directory.

Synopsis

```
procedure remove_temp_directory()
```

Further information

1. Can only be called when no model is being executed and if the default working directory is not a subdirectory of the temporary directory.
2. The temporary directory is also automatically removed when Mosel is finalized (at module unload).

Module

moselpy

CHAPTER 8

I/O Driver

This package provides the `moselpy:` I/O driver at the Mosel language level.

When used without a label, the `moselpy:` I/O driver must be employed as a file name for initializations blocks for bidirectional data exchange between Python and Mosel models. Data passed to `Model.run` becomes available to the model via `initializations from "moselpy:"`, and data written via `initializations to "moselpy:"` is returned by `Model.run`.

When specifying a label (*e.g.* `moselpy:name`), the driver acts as a data stream (binary or textual depending on the associated Python entity) that may appear wherever a file name is expected. In particular, it is used for redirecting default input/output streams of Mosel (see `Model.set_default_stream`).

8.1 Driver *moselpy* without a label

The driver exchanges data by identifier name. Input data is provided as a dictionary to `Model.run`:

```
# Python
model = mp.load_model("model.bim")
output = model.run(input_data={
    "scalar_int": 42,
    "scalar_real": 3.14,
    "scalar_str": "hello",
    "array_1d": {1: 10, 2: 20, 3: 30},
    "array_2d": {(1,1): 100, (1,2): 200, (2,1): 300, (2,2): 400, (3,1): 500, (3,2): 600}
})
print(output["myresult"])
```

The corresponding Mosel model reads and writes using the driver:

```
model 'example'
declarations
  scalar_int: integer
  scalar_real: real
  scalar_str: string
  array_1d: array(R: range) of integer
  array_2d: array(R, C: range) of integer
  myresult: integer
end-declarations

! Explicit set specifications may be required to state the full index domain
initializations from "moselpy:"
  scalar_int scalar_real scalar_str array_1d array_2d
end-initializations

! Compute myresult using both arrays
myresult := scalar_int + sum(i in R) array_1d(i) + sum(i in R, j in C) array_2d(i, j)

! Send output to Python
```

```
initializations to "moselpy:"
  myresult
end-initializations
end-model
```

When arrays are passed from Python, Mosel populates array index sets (ranges and sets) automatically from the keys of the incoming data. Explicit initialization of R or C from Python is not needed. Explicit set specifications are only required for sparse arrays where the full index domain must be declared independently of the data.

8.1.1 Type conversion

The following type conversions are performed automatically:

Python Type	Mosel Type	Notes
`int`	`integer`	
`float`	`real`	
`str`	`string`	
`bool`	`boolean`	
`dict`	`array`	Keys become indices
`list`, `tuple`, `set`	`set` or `list`	Determined by Mosel declaration
`bytes`	Raw bytes	For binary driver
pandas `Series`	`array`	Index becomes array key; name used as identifier if not specified
pandas `DataFrame`	`array(row, col)`	Row index and column name form a 2-tuple key
NumPy `ndarray`	`array`	Integer-indexed from 1

For arrays, dictionary keys can be:

- Integers for single-index arrays: `{1: 10, 2: 20}` (NumPy integer types such as `np.int64` are also accepted)
- Tuples for multi-index arrays: `{(1,1): 100, (1,2): 200}` (tuples of NumPy integers are also accepted)
- Strings for string-indexed arrays: `{"a": 10, "b": 20}`

For Mosel `set` and `list` types, any Python iterable (list, tuple, set, frozenset) can be used. The target type is determined by the Mosel declaration, not the Python type.

8.1.2 Sparse arrays

Sparse arrays are supported naturally since Python dictionaries only contain defined entries:

```
# Python - sparse data (only some indices defined)
model.run(input_data={
    "sparse_arr": {1: 100, 5: 500, 10: 1000}
})
```

```
! Mosel - receives sparse array
declarations
  sparse_arr: dynamic array(range) of integer
end-declarations
```

```
initializations from "moselpy:"
  sparse_arr  ! Only indices 1, 5, 10 will be set
end-initializations
```

8.2 Driver *moselpy* with a label

The label associated to the `moselpy:` driver must have been registered in the `symbols` registry (see `Model.symbols`).

Example generating a binary data file in memory (see `to_binary`):

```
# Python
bin_data = mp.to_binary({"x": 42, "arr": {1: 10, 2: 20}})
model.symbols["mydata"] = bin_data
model.run()
```

```
! Mosel
initializations from "bin:moselpy:mydata"
  x arr
end-initializations
```

Example using a text string as a model source:

```
# Python
from io import StringIO

model_source = """model hello
  writeln('hello world')
end-model
"""
mp.symbols["mosel_source"] = StringIO(model_source)
mp.execute_model("moselpy:mosel_source")
```

CHAPTER 9

Status Classes

These classes provide detailed status information about optimization problems and model security. They are accessed via `Model.prob_status` and `Model.sec_status` properties.

9.1 ProblemStatus class

`ProblemStatus` Provides information about the optimization problem status. p. 66

`ProblemStatus.is_optimization_failed` Whether optimization failed. p. 74

`ProblemStatus.is_optimization_finished` Whether optimization has finished (regardless of outcome). p. 71

`ProblemStatus.is_problem_infeasible` Whether the problem is infeasible. p. 72

`ProblemStatus.is_problem_unbounded` Whether the problem is unbounded. p. 73

`ProblemStatus.is_problem_valid` Whether the current problem matrix is valid. p. 68

`ProblemStatus.is_solution_available` Whether a solution is available. p. 69

`ProblemStatus.is_solution_optimal` Whether the solution is optimal. p. 70

`ProblemStatus.status_code` Raw status code (for advanced use). p. 67

ProblemStatus

Purpose

Provides information about the optimization problem status.

Synopsis

```
class ProblemStatus
```

Further information

Accessed via `Model.prob_status` (or its alias `problem_status`). Cannot be instantiated directly.

Module

moselpy

ProblemStatus.status_code

Purpose

Raw status code (for advanced use).

Synopsis

```
prob_status.status_code -> int
```

Return value

Integer status code. Returns -1 if the model has not been run, 0 if no problem is available.

Further information

For checking optimization results, use the boolean properties instead. This property exposes the raw bit-encoded value for cases requiring direct access to the underlying status flags.

Module

moselpy

ProblemStatus.is_problem_valid

Purpose

Whether the current problem matrix is valid.

Synopsis

```
prob_status.is_problem_valid -> bool
```

Return value

`True` if the problem loaded in the optimizer is still valid

Module

moselpy

ProblemStatus.is_solution_available

Purpose

Whether a solution is available.

Synopsis

```
prob_status.is_solution_available -> bool
```

Return value

`True` if optimization has produced a solution

Module

moselpy

ProblemStatus.is_solution_optimal

Purpose

Whether the solution is optimal.

Synopsis

```
prob_status.is_solution_optimal -> bool
```

Return value

`True` if an optimal solution was found

Module

moselpy

ProblemStatus.is_optimization_finished

Purpose

Whether optimization has finished (regardless of outcome).

Synopsis

```
prob_status.is_optimization_finished -> bool
```

Return value

`True` if the optimizer has finished (solved, interrupted, or failed)

Module

moselpy

ProblemStatus.is_problem_infeasible

Purpose

Whether the problem is infeasible.

Synopsis

```
prob_status.is_problem_infeasible -> bool
```

Return value

`True` if the problem has no feasible solution

Module

moselpy

ProblemStatus.is_problem_unbounded

Purpose

Whether the problem is unbounded.

Synopsis

```
prob_status.is_problem_unbounded -> bool
```

Return value

`True` if the problem has unbounded objective

Module

moselpy

ProblemStatus.is_optimization_failed

Purpose

Whether optimization failed.

Synopsis

```
prob_status.is_optimization_failed -> bool
```

Return value

`True` if optimization terminated due to an error

Module

moselpy

9.2 SecurityStatus class

`SecurityStatus` Provides information about model signing and encryption status. p. 76

`SecurityStatus.is_encrypted` Whether the model is encrypted. p. 79

`SecurityStatus.is_signature_checked` Whether the signature has been verified. p. 80

`SecurityStatus.is_signature_verified` Whether the signature was successfully verified. p. 81

`SecurityStatus.is_signed` Whether the model has a digital signature. p. 78

`SecurityStatus.status_code` Raw security status code. p. 77

SecurityStatus

Purpose

Provides information about model signing and encryption status.

Synopsis

```
class SecurityStatus
```

Further information

Accessed via `Model.sec_status`. Cannot be instantiated directly.

Module

moselpy

SecurityStatus.status_code

Purpose

Raw security status code.

Synopsis

```
sec_status.status_code -> int
```

Return value

Integer status code with security flags

Module

moselpy

SecurityStatus.is_signed

Purpose

Whether the model has a digital signature.

Synopsis

```
sec_status.is_signed -> bool
```

Return value

`True` if the model was signed during compilation

Related topics

`compile_model`

Module

moselpy

SecurityStatus.is_encrypted

Purpose

Whether the model is encrypted.

Synopsis

```
sec_status.is_encrypted -> bool
```

Return value

`True` if the model was encrypted during compilation

Module

moselpy

SecurityStatus.is_signature_checked

Purpose

Whether the signature has been verified.

Synopsis

```
sec_status.is_signature_checked -> bool
```

Return value

`True` if signature verification was attempted

Module

moselpy

SecurityStatus.is_signature_verified

Purpose

Whether the signature was successfully verified.

Synopsis

```
sec_status.is_signature_verified -> bool
```

Return value

`True` if the signature is valid

Module

moselpy

Chapter 10

Enumerations

The package defines several enumeration types for status codes and flags.

10.1 ExecStatus enumeration

ExecStatus

Purpose

Execution status codes returned by `Model.exec_status`.

Synopsis

```
enum ExecStatus
```

Arguments

`OK`	Normal termination
`MATHERR`	Mathematical error (*e.g.*, division by zero)
`ERROR`	An error occurred during execution
`IOERR`	Input/output error (*e.g.*, cannot open file)
`BREAK`	Execution interrupted by breakpoint (debugger)
`NIFCT`	Native interface function error
`NULLREF`	Null reference error occurred
`LICERR`	Execution could not start due to license error
`STOP`	Bit set if execution was interrupted by `Model.stop`

Example

Check execution result:

```
model.run()
if model.exec_status == mp.ExecStatus.OK:
    print("Model executed successfully")
elif model.exec_status == mp.ExecStatus.ERROR:
    print("Runtime error occurred")
```

Further information

Note that STOP is a bit that can be combined with other status values. Check with bitwise AND if needed: `status & mp.ExecStatus.STOP`.

Related topics

`Model.exec_status`

Module

moselpy

10.2 StreamType enumeration

StreamType

Purpose

Stream type flags for `Config.set_default_stream` and `Model.set_default_stream`.

Synopsis

```
enum StreamType (flags)
```

Arguments

`INPUT`	Input stream
`OUTPUT`	Output stream
`APPEND`	Append mode (can be combined with OUTPUT)
`ERROR`	Error stream
`LINBUF`	Line buffered mode (can be combined with OUTPUT)

Further information

This is a flag enumeration; values can be combined using bitwise OR.

Related topics

`Config.set_default_stream`, `Model.set_default_stream`

Module

moselpy

CHAPTER 11

Binary Format Utilities

These functions work with Mosel's binary data format, which is used for efficient data exchange via the `bin:` I/O driver.

`from_binary`	Parse Mosel binary format bytes to a Python dictionary.	p. 88
`read_binary`	Read a file in Mosel binary format to a Python dictionary.	p. 90
`to_binary`	Convert a Python dictionary to Mosel binary format bytes.	p. 87
`write_binary`	Write a Python dictionary to a file in Mosel binary format.	p. 89

to_binary

Purpose

Convert a Python dictionary to Mosel binary format bytes.

Synopsis

```
function to_binary(data) -> bytes
```

Argument

`data` Dictionary with string keys and values convertible to Mosel types

Return value

Binary data as a `bytes` object

Example

Create binary data for a Mosel model:

```
data = {"x": 42, "arr": {1: 10, 2: 20}}
bin_data = mp.to_binary(data)
output = model.run(input_data={"mydata": bin_data})
```

Further information

1. The binary format is the same as used by Mosel's `bin:` driver. In the Mosel model, use `initializations from "bin:moselpy:name"` to read the binary data.
2. Binary format is more efficient than text format for large datasets and preserves full precision for floating-point numbers.

Related topics

`from_binary`, `write_binary`

Module

moselpy

from_binary

Purpose

Parse Mosel binary format bytes to a Python dictionary.

Synopsis

```
function from_binary(data) -> dict
```

Argument

`data` Binary data as a `bytes` object

Return value

Dictionary with decoded values

Example

Parse binary output from a model:

```
output = model.run()
result = mp.from_binary(output["binary_result"])
print(result)
```

Further information

Use this to decode data written by a Mosel model using `initializations to "bin:moselpy:name"`.

Related topics

`to_binary`, `read_binary`

Module

moselpy

write_binary

Purpose

Write a Python dictionary to a file in Mosel binary format.

Synopsis

```
procedure write_binary(path, data)
```

Arguments

`path` Output file path

`data` Dictionary to write

Example

Write data to a binary file:

```
mp.write_binary("data.bin", {"x": 42, "arr": {1: 10, 2: 20}})
```

Further information

Creates a file that can be read by Mosel using the `bin:` driver: `initializations from "bin:data.bin"`

Related topics

`read_binary`, `to_binary`

Module

moselpy

read_binary

Purpose

Read a file in Mosel binary format to a Python dictionary.

Synopsis

```
function read_binary(path) -> dict
```

Argument

`path` Input file path

Return value

Dictionary with decoded values

Example

Read binary file created by Mosel:

```
data = mp.read_binary("output.bin")
print(data)
```

Further information

Reads files created by Mosel's `bin:` driver: `initializations to "bin:output.bin"`

Related topics

`write_binary`, `from_binary`

Module

moselpy

Appendix A

Contacting FICO

FICO provides clients with support and services for all our products.

FICO Customer Support

FICO Customer Support offers technical support and services ranging from self-help tools to direct assistance with a FICO technical support engineer. Support is available to all clients who have an active maintenance contract.

The FICO Customer Self-Service Portal (support.fico.com) is a secure web portal that allows users to open, review, and update their support cases; manage their organization's portal users; find solutions to common problems in the FICO Knowledge Base; and view the availability of their cloud applications 24 hours a day, 7 days a week.

You can find support contact information and a link to the FICO Customer Self-Service Portal (online support) on the Product Support home page (www.fico.com/en/product-support).

Please include *'Xpress'* in the subject line of your support queries.

Documentation

FICO continually looks for new ways to improve and enhance the value of the products and services we provide.

If you have comments or suggestions regarding how we can improve this documentation, let us know by sending your suggestions to techpubs@fico.com. Please include your contact information (name, company, email address, and optionally, your phone number) so we may reach you if we have questions.

FICO Learning

FICO Learning is the principal provider of product training for our clients and partners. FICO Learning offers instructor-led classroom courses, web-based training, seminars, and training tools for both new user enablement and ongoing performance support.

For additional information, visit the FICO Learning home page at www.fico.com/en/product-training or email producteducation@fico.com.

Sales and maintenance

If you need information on other Xpress Optimization products, or you need to discuss maintenance contracts or other sales-related items, contact FICO by:

- Phone: +1 (408) 535-1500 or +44 207 940 8718
- Web: www.fico.com/optimization and use the available contact forms

About FICO

FICO (NYSE:FICO) is a leading analytics software company, helping businesses in 90+ countries make better decisions that drive higher levels of growth, profitability, and customer satisfaction. Learn more at www.fico.com or contact us at www.fico.com/en/contact-us.

Index

B
`bim_prefix`, 27
`bim_version`, 30

C
compile
    model, 58
`compile_model`, 58
compile_model, 58
`Config`, 26
`config`, 30
`Config.set_default_stream`, 29

D
`default_working_directory`, 27

E
exec_status, 45
`ExecStatus`, 83
execute
    model, 34
`execute_model`, 60
execute_model, 60
execution status, 83
exit_code, 46
export_problem, 39

F
find_identifier, 35
`from_binary`, 88
from_binary, 88

G
get_annotations, 40

I
I/O driver
    moselpy:, 62
identifier
    find, 35
is_encrypted, 79
is_loaded, 43
is_optimization_failed, 74
is_optimization_finished, 71
is_problem_infeasible, 72
is_problem_unbounded, 73
is_problem_valid, 68
is_running, 44
is_signature_checked, 80
is_signature_verified, 81
is_signed, 78
is_solution_available, 69
is_solution_optimal, 70

K
key_fingerprint, 50

L
`library_path`, 30
load
    model, 59
`load_model`, 59
load_model, 59
`locale_directory`, 27

M
`max_stack_dump_depth`, 27
`Model`, 33
model
    compile, 58
    load, 59
    reset, 38
    run, 34
`Model.exec_status`, 45
`Model.exit_code`, 46
`Model.export_problem`, 39
`Model.find_identifier`, 35
`Model.get_annotations`, 40
`Model.is_loaded`, 43
`Model.is_running`, 44
`Model.key_fingerprint`, 50
`Model.name`, 41
`Model.number`, 42
`Model.objective_value`, 47
`Model.prob_status`, 48
`Model.reset`, 38
`Model.run`, 34
`Model.sec_status`, 49
`Model.set_default_stream`, 56
`Model.size`, 51
`Model.stop`, 36
`Model.symbols`, 53
`Model.sys_comment`, 54
`Model.terminate`, 37
`Model.user_comment`, 55
`Model.version`, 52
`mosel_version`, 30
`moselpy:`, 62

N
name
    Model, 41
`native_version`, 30
number
    Model, 42

O
objective_value, 47

P
prob_status, 48
problem
    export, 39
    status, 66
`ProblemStatus`, 66
`ProblemStatus.is_optimization_failed`, 74
`ProblemStatus.is_optimization_finished`, 71
`ProblemStatus.is_problem_infeasible`, 72
`ProblemStatus.is_problem_unbounded`, 73
`ProblemStatus.is_problem_valid`, 68
`ProblemStatus.is_solution_available`, 69
`ProblemStatus.is_solution_optimal`, 70
`ProblemStatus.status_code`, 67

R
`read_binary`, 90
read_binary, 90
`remove_temp_directory`, 61
remove_temp_directory, 61
reset
    Model, 38
run
    Model, 34

S
sec_status, 49
`SecurityStatus`, 76
`SecurityStatus.is_encrypted`, 79
`SecurityStatus.is_signature_checked`, 80
`SecurityStatus.is_signature_verified`, 81
`SecurityStatus.is_signed`, 78
`SecurityStatus.status_code`, 77
set_default_stream
    Config, 29
    Model, 56
size
    Model, 51
status_code
    ProblemStatus, 67
    SecurityStatus, 77
stop
    Model, 36
`StreamType`, 85
symbols
    Model, 53
`symbols`, 31
sys_comment, 54

T
temporary directory, 61
terminate
    Model, 37
`to_binary`, 87
to_binary, 87

U
user_comment, 55

V
version
    Model, 52
`version`, 31

W
`write_binary`, 89
write_binary, 89

X
`xpress_version`, 31