informs

ANALYTICS

CONFERENCE

informs
ANALYTICS
CONFERENCE

GAMSPy: Algebraic Modeling in Python

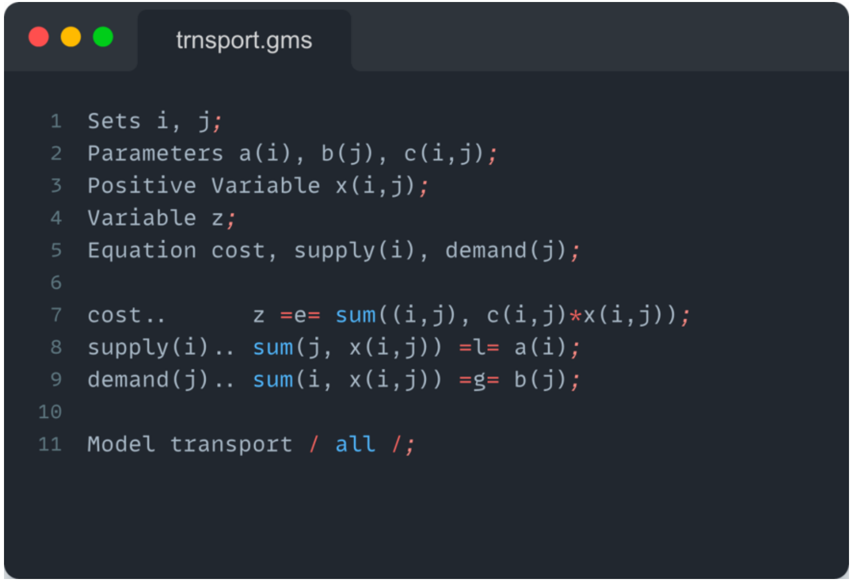
What is GAMS?...

- GAMS is a specialized, high performance, algebraic modeling language (AML)

- Active development since 1987

- Backward compatibility is foundational to our mission

What is GAMS?

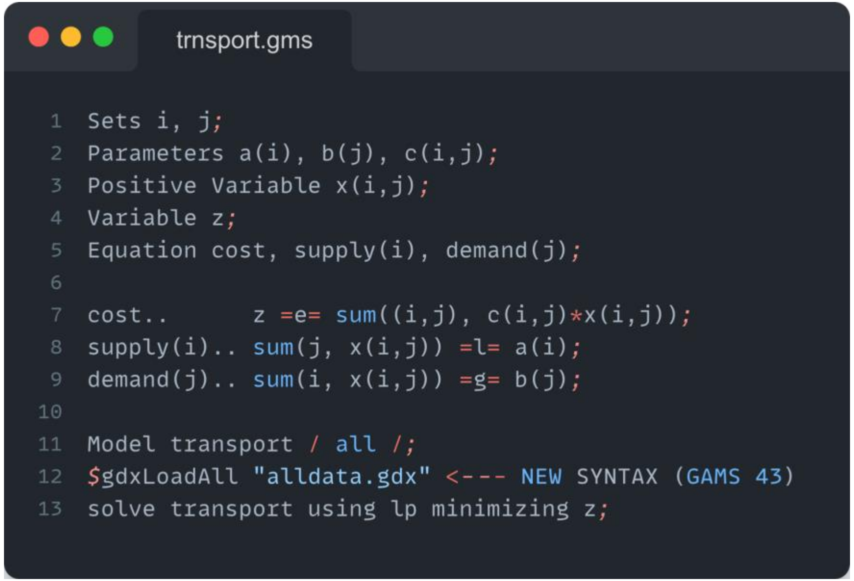


```
Sets i, j;
Parameters a(i), b(j), c(i,j);
Positive Variable x(i,j);
Variable z;
Equation cost, supply(i), demand(j);

cost..      z =e= sum((i,j), c(i,j)*x(i,j));
supply(i).. sum(j, x(i,j)) =l= a(i);
demand(j).. sum(i, x(i,j)) =g= b(j);

Model transport / all /;
```

Build abstract models

- Declared/defined over sets
- Like “writing on paper”
- Compact
- Logically consistent
  - no domain violations
  - no uncontrolled sets
- Completely abstract (no data)

What is GAMS?



```
Sets i, j;
Parameters a(i), b(j), c(i,j);
Positive Variable x(i,j);
Variable z;
Equation cost, supply(i), demand(j);

cost..      z =e= sum((i,j), c(i,j)*x(i,j));
supply(i).. sum(j, x(i,j)) =l= a(i);
demand(j).. sum(i, x(i,j)) =g= b(j);

Model transport / all /;
$gdxLoadAll "alldata.gdx" <--- NEW SYNTAX (GAMS 43)
solve transport using lp minimizing z;
```

Model Instance

- Add all the data… fast! 🔥 🚒
- Model instance generated at solve

GAMS Philosophy – tight syntax leads to better modeling

The GAMS System – The Good

Algebraic Modeling Language (AML)

Backward Compatibility – **stable!**

Active development since 1987 – **stable!**

Familiar syntax – **human readable!**

GAMS data structures – **fast! large models!**

The GAMS System – The Could Be Better

Not a general programming language

Lacks modern look/feel to syntax

Learning curve issues, users want familiar syntax/reuse knowledge

Not easily deployable (~2GB install)

Integration into pipelines could be smoother

Relational data based (not general like `list`, `dict`, etc.)

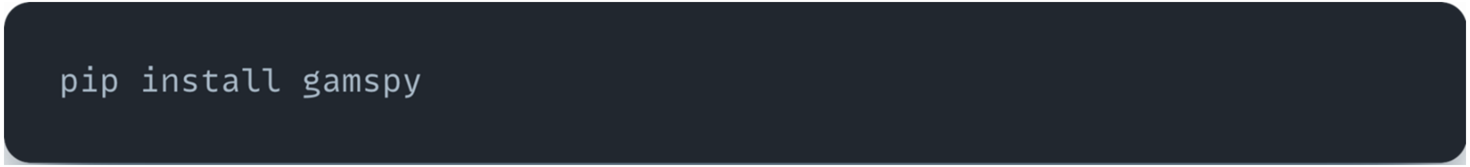
What is GAMSPy?...

- Python-based Algebraic Modeling Language (AML)
- Abstract/data-independent modeling
- Convenient handling of sparse data
- Streamlined optimization pipeline management
- Convenient and efficient data structures (Numpy, Pandas)
- Runs with a specialized engine (GAMS)

NumPy pandas

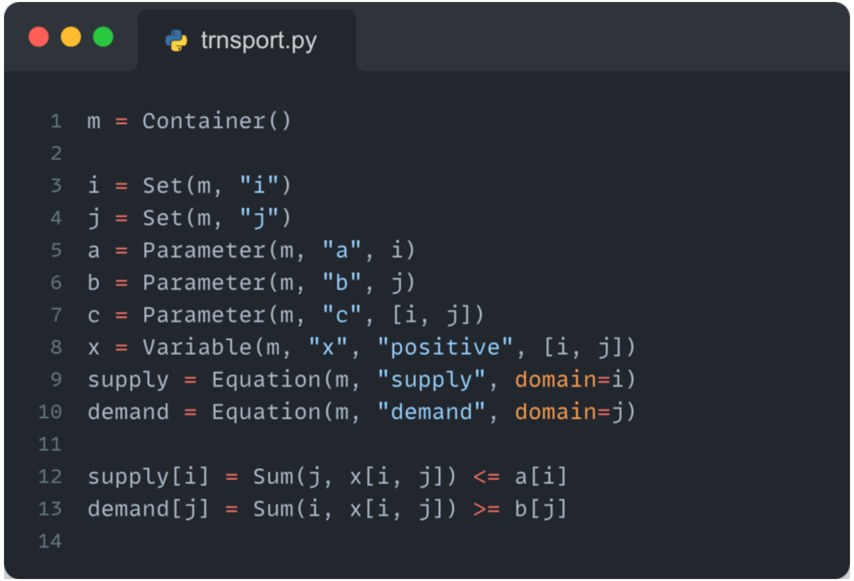
What is GAMSPy?...

- More Python-ic dev cycle
  - `FutureWarning`, release notes, `DeprecationWarning`, removal
- Currently in beta - 1.0 released in the next 4 months
- Documentation: **`gamspy.readthedocs.io`**
- Model library (100+ models): **`gamspy.readthedocs.io/en/latest/user/model_library.html`**



```
pip install gamspy
```

What is GAMSPy?



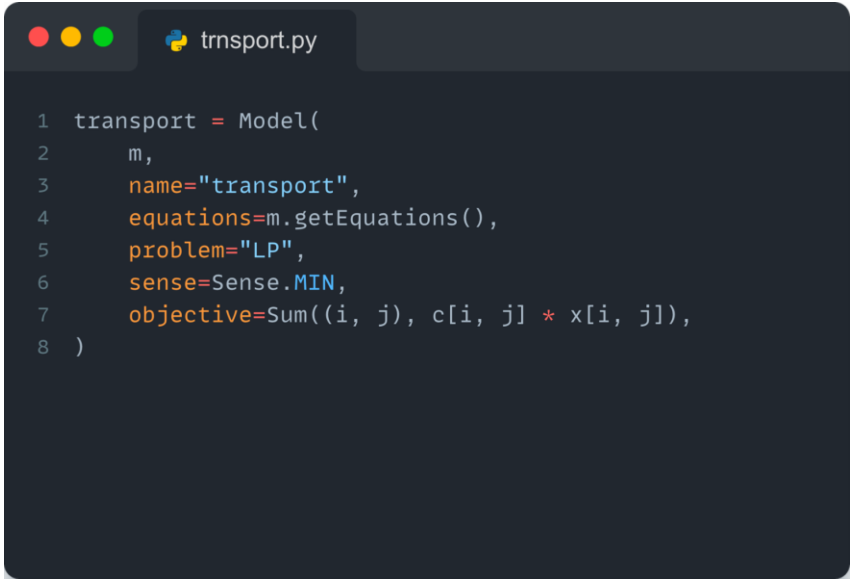
trnsport.py

```python
m = Container()

i = Set(m, "i")
j = Set(m, "j")
a = Parameter(m, "a", i)
b = Parameter(m, "b", j)
c = Parameter(m, "c", [i, j])
x = Variable(m, "x", "positive", [i, j])
supply = Equation(m, "supply", domain=i)
demand = Equation(m, "demand", domain=j)

supply[i] = Sum(j, x[i, j]) <= a[i]
demand[j] = Sum(i, x[i, j]) >= b[j]

```

Build abstract models

- Declared/defined over sets
- Like “writing on paper”
- Compact
- Logically consistent
  - no domain violations
  - no uncontrolled sets
- Completely abstract (no data)
- *Leverages operator overloading*

What is GAMSPy?

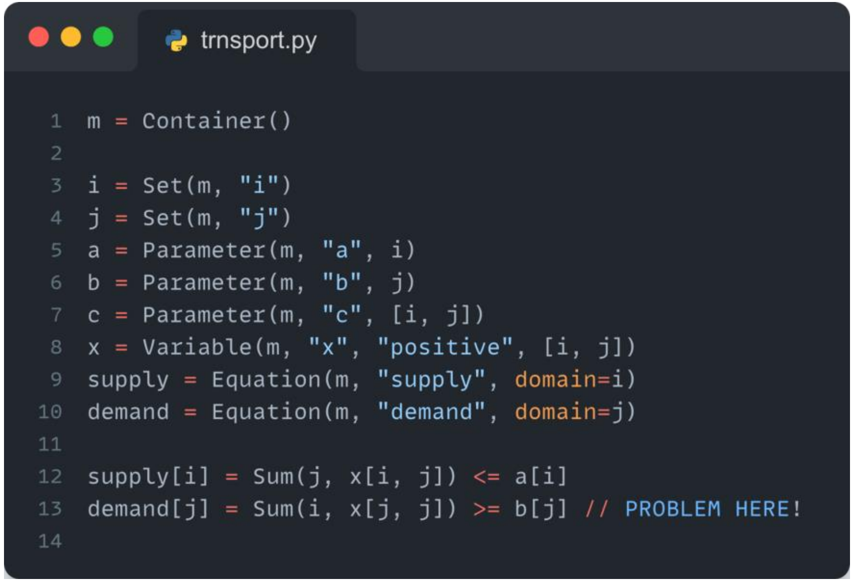


```
transport = Model(
    m,
    name="transport",
    equations=m.getEquations(),
    problem="LP",
    sense=Sense.MIN,
    objective=Sum((i, j), c[i, j] * x[i, j]),
)
```

Build abstract models

- Declared/defined over sets
- Like “writing on paper”
- Compact
- Logically consistent
  - no domain violations
  - no uncontrolled sets
- Completely abstract (no data)
- *Leverages operator overloading*

What is GAMSPy?

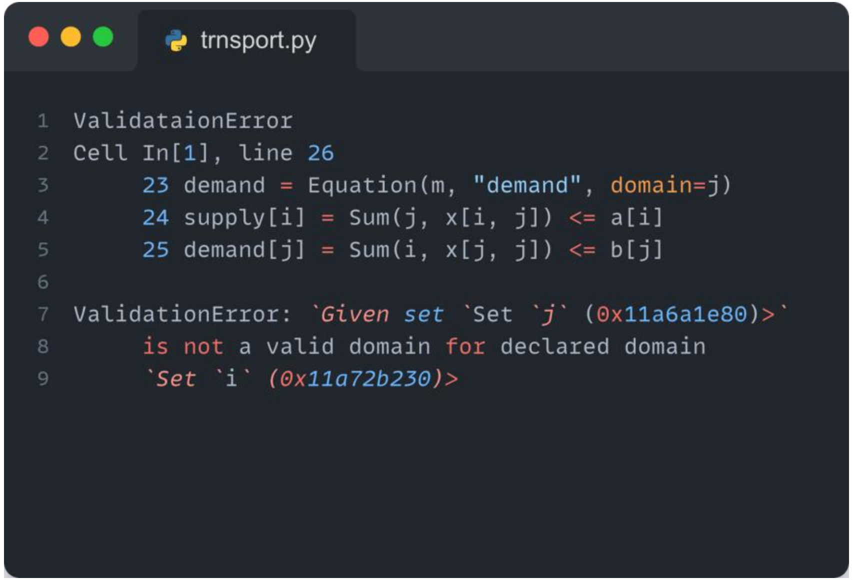


```python
m = Container()

i = Set(m, "i")
j = Set(m, "j")
a = Parameter(m, "a", i)
b = Parameter(m, "b", j)
c = Parameter(m, "c", [i, j])
x = Variable(m, "x", "positive", [i, j])
supply = Equation(m, "supply", domain=i)
demand = Equation(m, "demand", domain=j)

supply[i] = Sum(j, x[i, j]) <= a[i]
demand[j] = Sum(i, x[j, j]) >= b[j] // PROBLEM HERE!

```

Build abstract models

- Declared/defined over sets
- Like “writing on paper”
- Compact
- Logically consistent
  - no domain violations
  - no uncontrolled sets
- Completely abstract (no data)
- ***Leverages operator overloading***

What is GAMSPy?



```
ValidataionError
Cell In[1], line 26
     23 demand = Equation(m, "demand", domain=j)
     24 supply[i] = Sum(j, x[i, j]) <= a[i]
     25 demand[j] = Sum(i, x[j, j]) <= b[j]

ValidationError: `Given set `Set `j` (0x11a6a1e80)>`
     is not a valid domain for declared domain
     `Set `i` (0x11a72b230)>
```

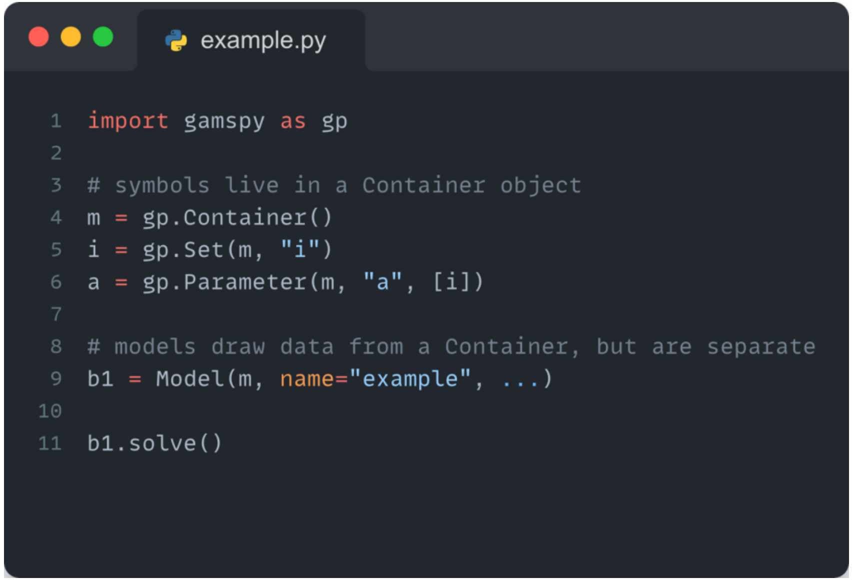
Generates Early Warnings

- Domain violations
- Cannot create equation block
- Raises helpful error messages

GAMS Philosophy – tight syntax leads to better modeling

🚀 Next… Getting Started! 🚀

GAMSPy Design Philosophy



example.py

```python
import gamspy as gp

# symbols live in a Container object
m = gp.Container()
i = gp.Set(m, "i")
a = gp.Parameter(m, "a", [i])

# models draw data from a Container, but are separate
b1 = Model(m, name="example", ...)

b1.solve()
```

… add symbols to a `Container`

… symbols get linked together (holding references)

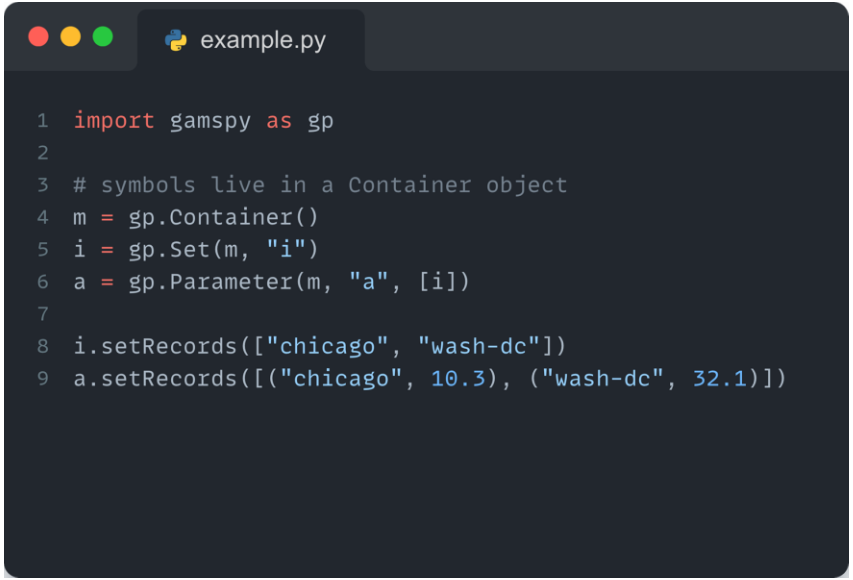
… `Models` are separate objects and do not live in a `Container`

… `Models` are solved, not `Containers`

GAMS

informs

GAMSPy Design Philosophy



```python
import gamspy as gp

# symbols live in a Container object
m = gp.Container()
i = gp.Set(m, "i")
a = gp.Parameter(m, "a", [i])

i.setRecords(["chicago", "wash-dc"])
a.setRecords([("chicago", 10.3), ("wash-dc", 32.1)])
```

… symbols hold `records` (as pandas DataFrames)

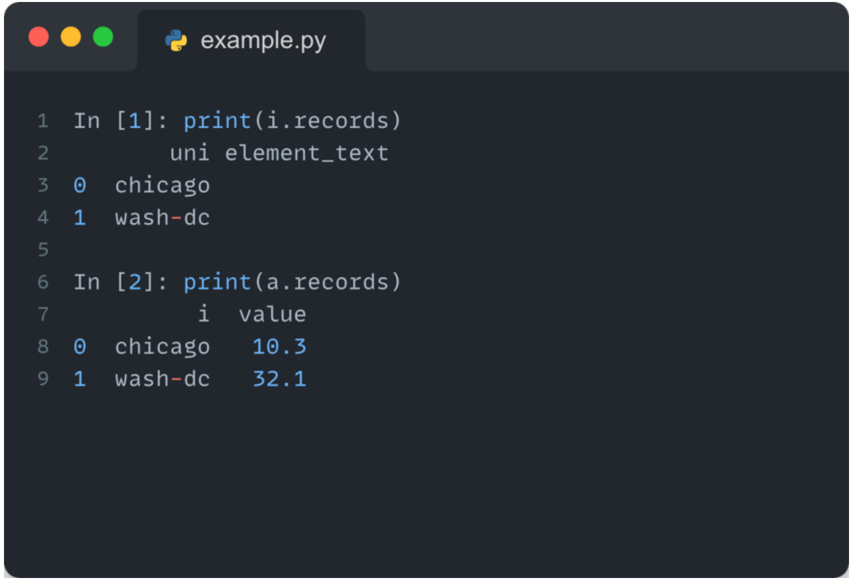
… symbol records are added with `setRecords` method

… many data types are accepted!

… `setRecords` will standardize the data

… can also pass data at symbol construction

GAMSPy Design Philosophy



```
In [1]: print(i.records)
       uni element_text
0  chicago
1  wash-dc

In [2]: print(a.records)
         i  value
0  chicago   10.3
1  wash-dc   32.1
```

… symbols hold `records` (as pandas DataFrames)

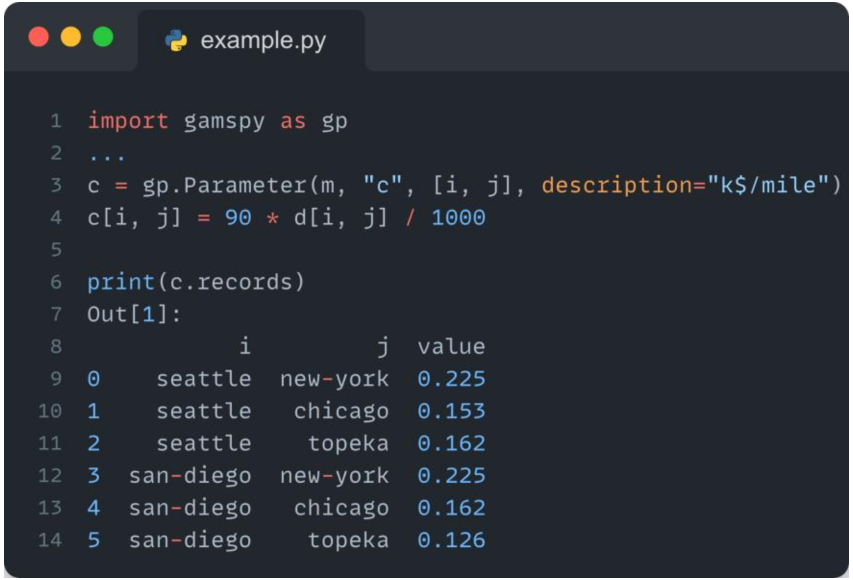
… symbol records are added with `setRecords` method

… many data types are accepted!

… `setRecords` will standardize the data

… can also pass data at symbol construction

GAMSPy Design Philosophy



```
import gamspy as gp
...
c = gp.Parameter(m, "c", [i, j], description="k$/mile")
c[i, j] = 90 * d[i, j] / 1000

print(c.records)
Out[1]:
            i         j  value
0     seattle  new-york  0.225
1     seattle   chicago  0.153
2     seattle    topeka  0.162
3   san-diego  new-york  0.225
4   san-diego   chicago  0.162
5   san-diego    topeka  0.126
```

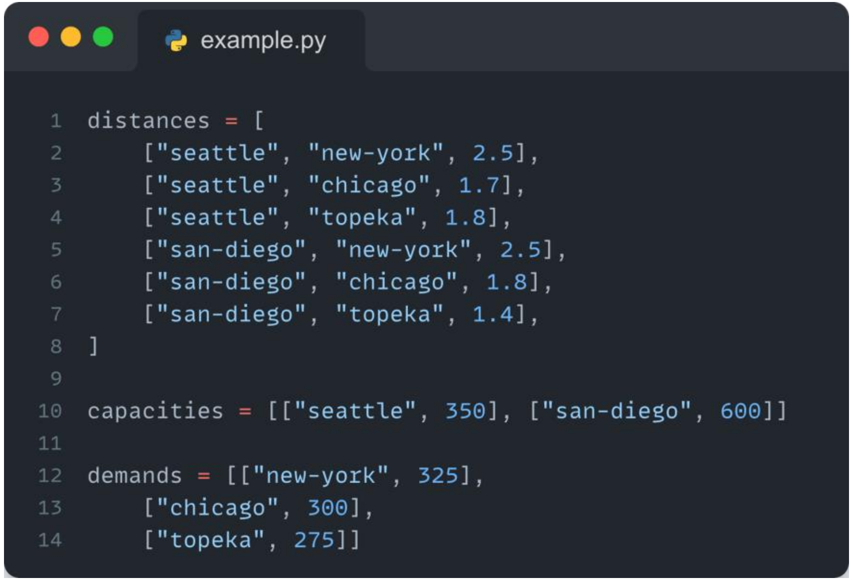
… algebra is added with familiar syntax (operator overloading)

… algebra is passed to GAMS subsystem and executed immediately

… results are available in Python

First Example – Prepare Data

… native python data types are OK



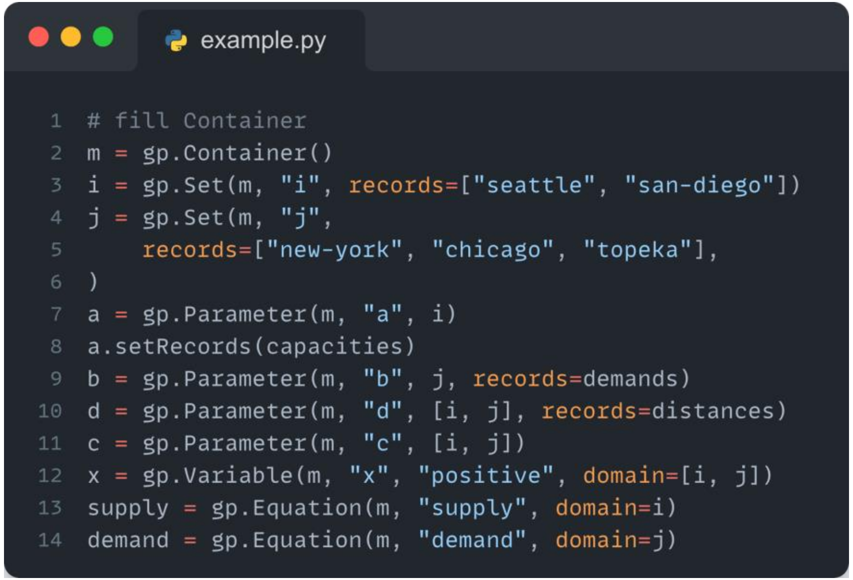
example.py

```python
distances = [
    ["seattle", "new-york", 2.5],
    ["seattle", "chicago", 1.7],
    ["seattle", "topeka", 1.8],
    ["san-diego", "new-york", 2.5],
    ["san-diego", "chicago", 1.8],
    ["san-diego", "topeka", 1.4],
]

capacities = [["seattle", 350], ["san-diego", 600]]

demands = [["new-york", 325],
    ["chicago", 300],
    ["topeka", 275]]
```

GAMS

informs

First Example – Fill the Container



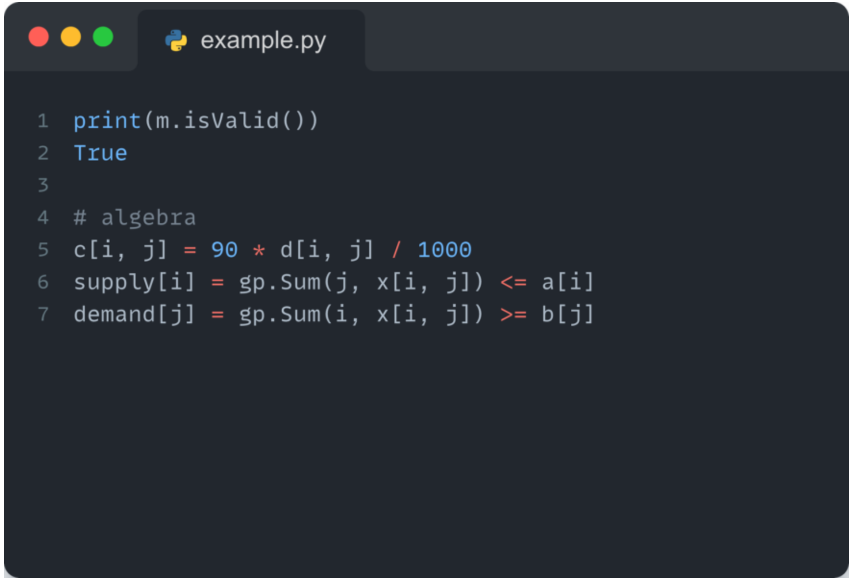
```python
# fill Container
m = gp.Container()
i = gp.Set(m, "i", records=["seattle", "san-diego"])
j = gp.Set(m, "j",
    records=["new-york", "chicago", "topeka"],
)
a = gp.Parameter(m, "a", i)
a.setRecords(capacities)
b = gp.Parameter(m, "b", j, records=demands)
d = gp.Parameter(m, "d", [i, j], records=distances)
c = gp.Parameter(m, "c", [i, j])
x = gp.Variable(m, "x", "positive", domain=[i, j])
supply = gp.Equation(m, "supply", domain=i)
demand = gp.Equation(m, "demand", domain=j)
```

… can use a mix of programming styles to set records `(constructor, setRecords)`

… validate (and debug) data with `<symbol>.isValid()` or `<container>.isValid()`

… `verbose=True` will output helpful error messages

First Example – Define Algebra

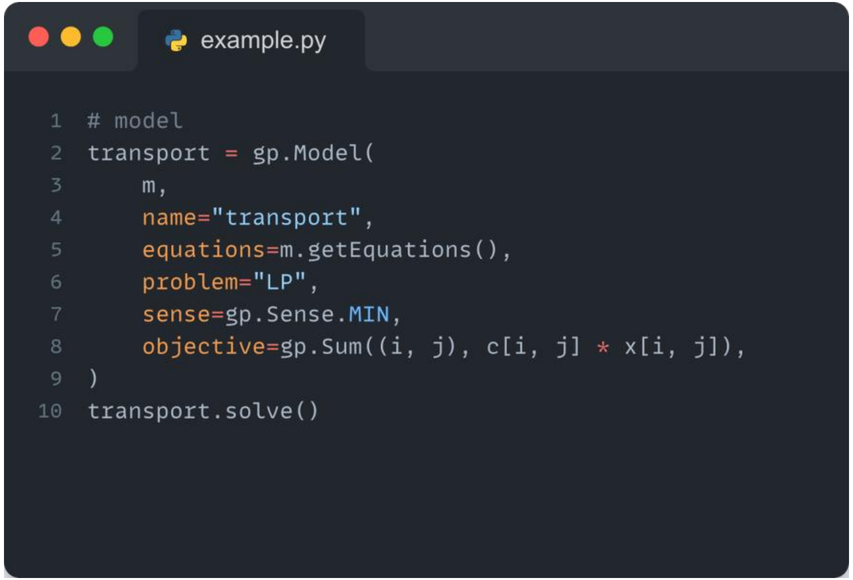


example.py

```python
print(m.isValid())
True

# algebra
c[i, j] = 90 * d[i, j] / 1000
supply[i] = gp.Sum(j, x[i, j]) <= a[i]
demand[j] = gp.Sum(i, x[i, j]) >= b[j]
```

… remember GAMSPy checks for logical inconsistencies in algebra (domain violations, uncontrolled sets)

First Example – Define Model and Solve



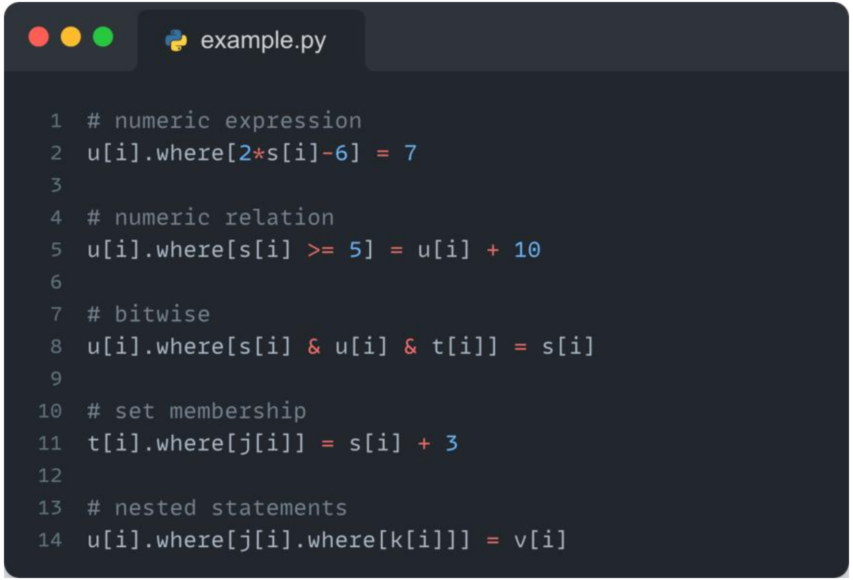
```python
# model
transport = gp.Model(
    m,
    name="transport",
    equations=m.getEquations(),
    problem="LP",
    sense=gp.Sense.MIN,
    objective=gp.Sum((i, j), c[i, j] * x[i, j]),
)
transport.solve()
```

… objective function algebra can be passed in `Model` constructor (same with `Equations`)

… `<model>.solve()` returns a DataFrame with results/status

… `Model` object holds a lot of metainformation about the model!

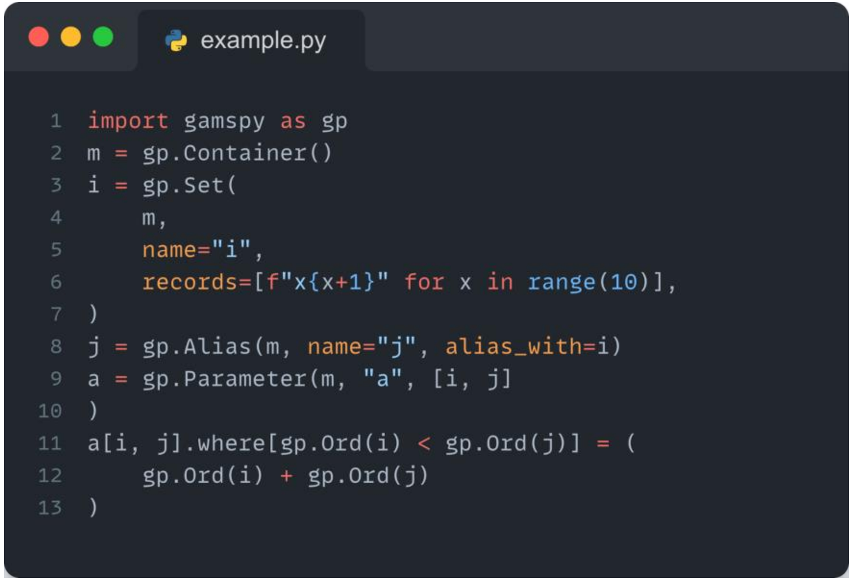
Next Level Syntax -- `.where`



```python
# numeric expression
u[i].where[2*s[i]-6] = 7

# numeric relation
u[i].where[s[i] >= 5] = u[i] + 10

# bitwise
u[i].where[s[i] & u[i] & t[i]] = s[i]

# set membership
t[i].where[j[i]] = s[i] + 3

# nested statements
u[i].where[j[i].where[k[i]]] = v[i]
```

- GAMS syntax relies on conditional statements for assignment `(Sets, Parameters, Equations)`
- GAMSPy supports
  - Numerical expressions
  - Numerical relations
  - Bitwise operations
  - Set membership
  - Mixed statements
  - Nested conditions
- *Works on LHS or RHS*

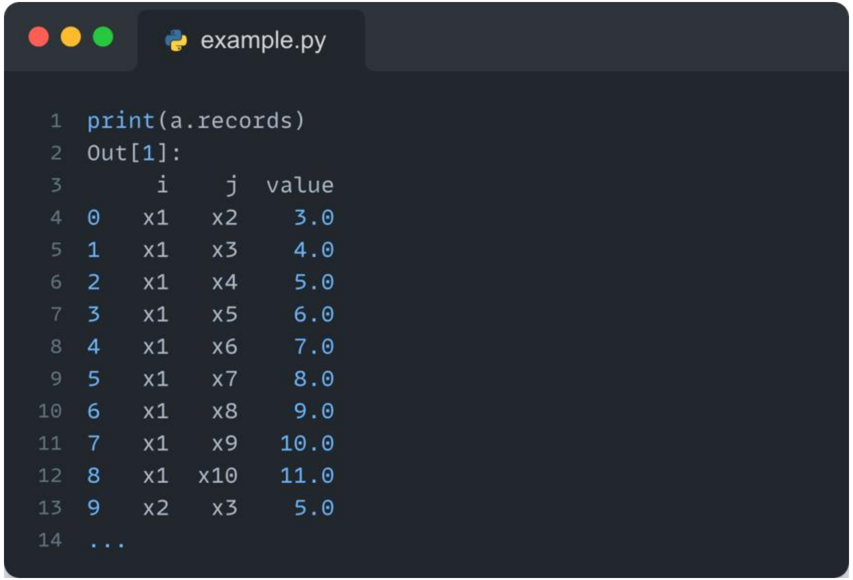
Other Advanced Syntax



```
import gamspy as gp
m = gp.Container()
i = gp.Set(
    m,
    name="i",
    records=[f"x{x+1}" for x in range(10)],
)
j = gp.Alias(m, name="j", alias_with=i)
a = gp.Parameter(m, "a", [i, j]
)
a[i, j].where[gp.Ord(i) < gp.Ord(j)] = (
    gp.Ord(i) + gp.Ord(j)
)
```

- `Ord` and `Card`
  - Useful with set to get positional/size information
- `Lag` and `Lead`
  - Relates set members (next or previous)
  - Linear or circular types

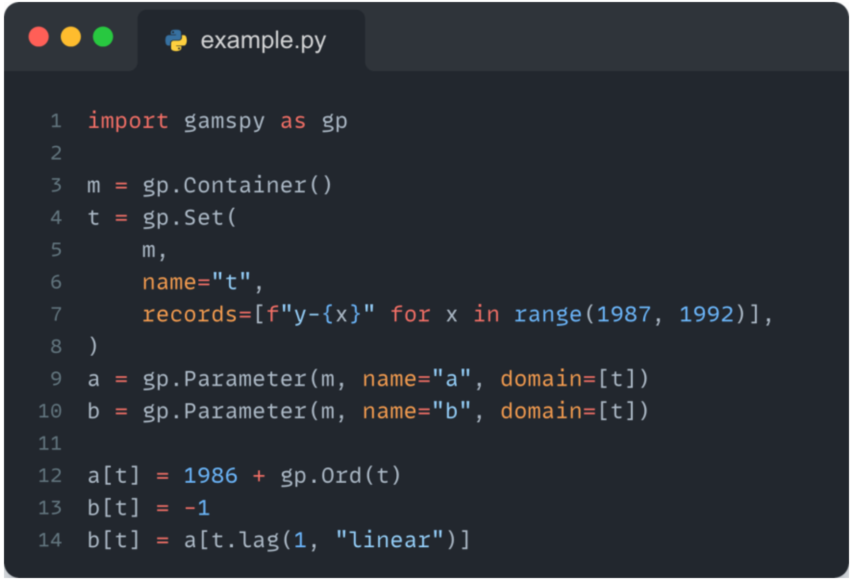
Other Advanced Syntax



```
print(a.records)
Out[1]:
     i    j  value
0   x1   x2    3.0
1   x1   x3    4.0
2   x1   x4    5.0
3   x1   x5    6.0
4   x1   x6    7.0
5   x1   x7    8.0
6   x1   x8    9.0
7   x1   x9   10.0
8   x1  x10   11.0
9   x2   x3    5.0
...
```

- `Ord` and `Card`
  - Useful with set to get positional/size information
- `Lag` and `Lead`
  - Relates set members (next or previous)
  - Linear or circular types

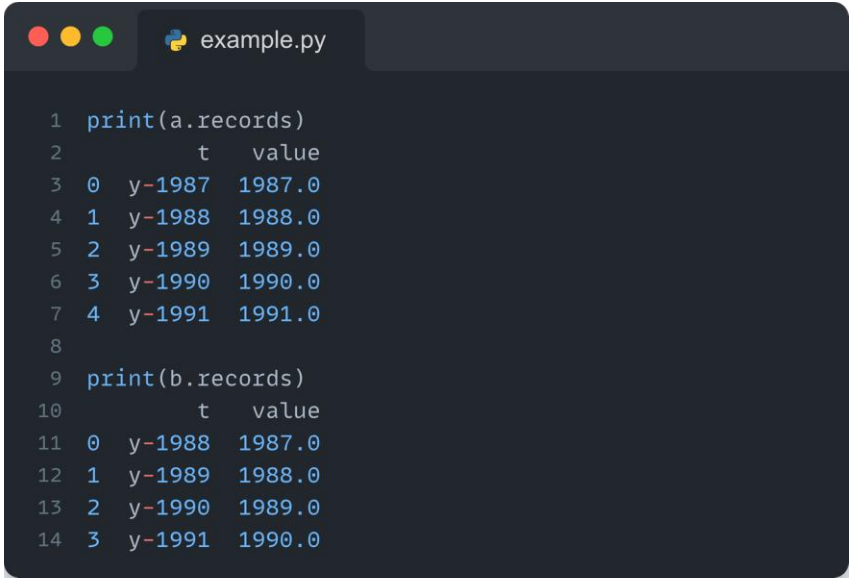
Other Advanced Syntax



```python
import gamspy as gp

m = gp.Container()
t = gp.Set(
    m,
    name="t",
    records=[f"y-{x}" for x in range(1987, 1992)],
)
a = gp.Parameter(m, name="a", domain=[t])
b = gp.Parameter(m, name="b", domain=[t])

a[t] = 1986 + gp.Ord(t)
b[t] = -1
b[t] = a[t.lag(1, "linear")]
```

- `Ord` and `Card`
  - Useful with set to get positional/size information
- `Lag` and `Lead`
  - Relates set members (next or previous)
  - Linear or circular types

Other Advanced Syntax



```
print(a.records)
           t   value
0   y-1987  1987.0
1   y-1988  1988.0
2   y-1989  1989.0
3   y-1990  1990.0
4   y-1991  1991.0

print(b.records)
           t   value
0   y-1988  1987.0
1   y-1989  1988.0
2   y-1990  1989.0
3   y-1991  1990.0
```

- Ord and Card
  - Useful with set to get positional/size information
- Lag and Lead
  - Relates set members (next or previous)
  - Linear or circular types

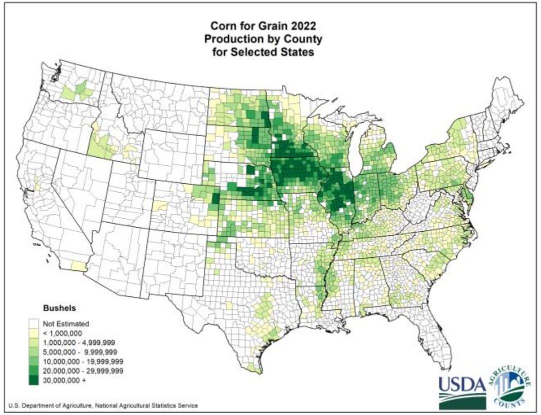
GAMS

informs

A Real Example…

- Uncertain data of corn production (county/state level)
- Confident of national level data
- Need balanced data
  - ∑counties = states
  - ∑states = nation

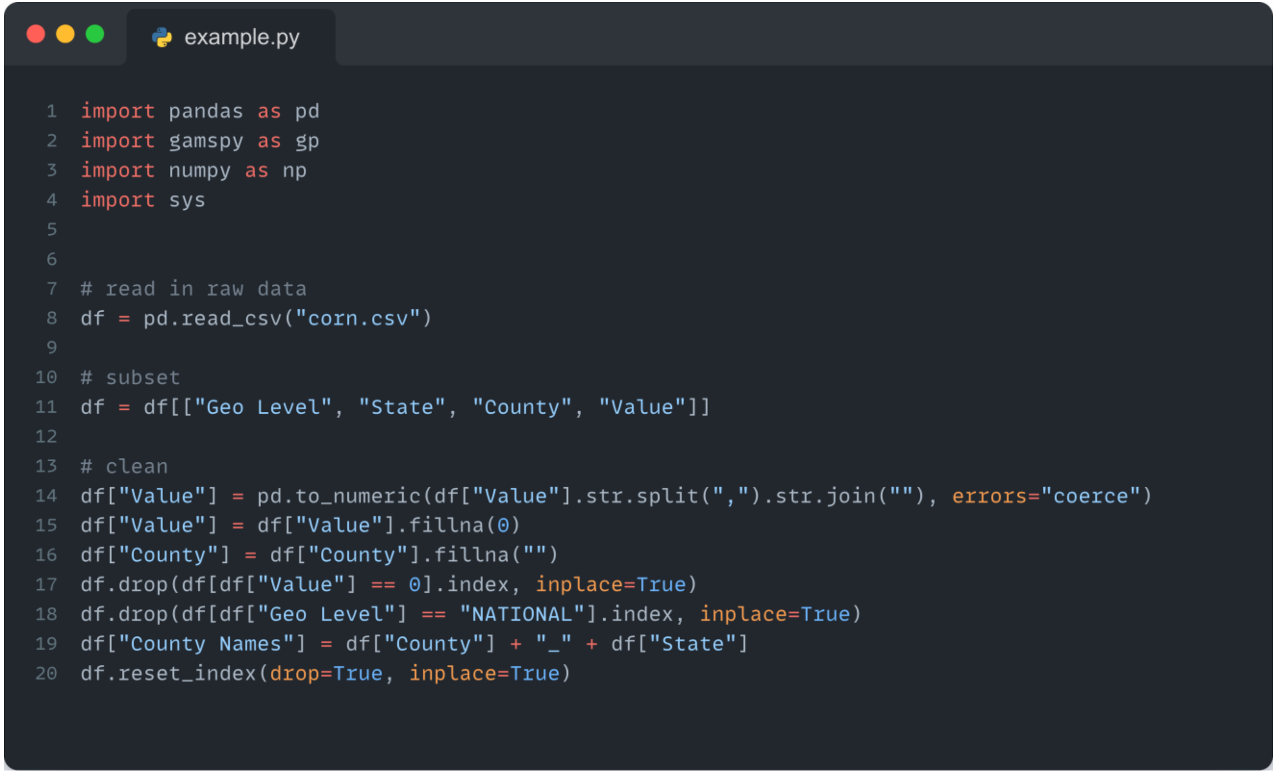
- Minimize weighted square error



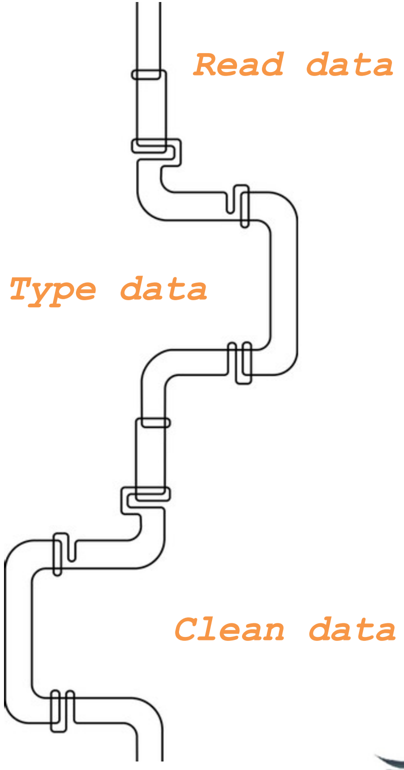
Corn for Grain 2022
Production by County
for Selected States

U.S. Department of Agriculture, National Agricultural Statistics Service

Single Environment Pipelines



```python
import pandas as pd
import gamspy as gp
import numpy as np
import sys


# read in raw data
df = pd.read_csv("corn.csv")

# subset
df = df[["Geo Level", "State", "County", "Value"]]

# clean
df["Value"] = pd.to_numeric(df["Value"].str.split(",").str.join(""), errors="coerce")
df["Value"] = df["Value"].fillna(0)
df["County"] = df["County"].fillna("")
df.drop(df[df["Value"] == 0].index, inplace=True)
df.drop(df[df["Geo Level"] == "NATIONAL"].index, inplace=True)
df["County Names"] = df["County"] + "_" + df["State"]
df.reset_index(drop=True, inplace=True)
```

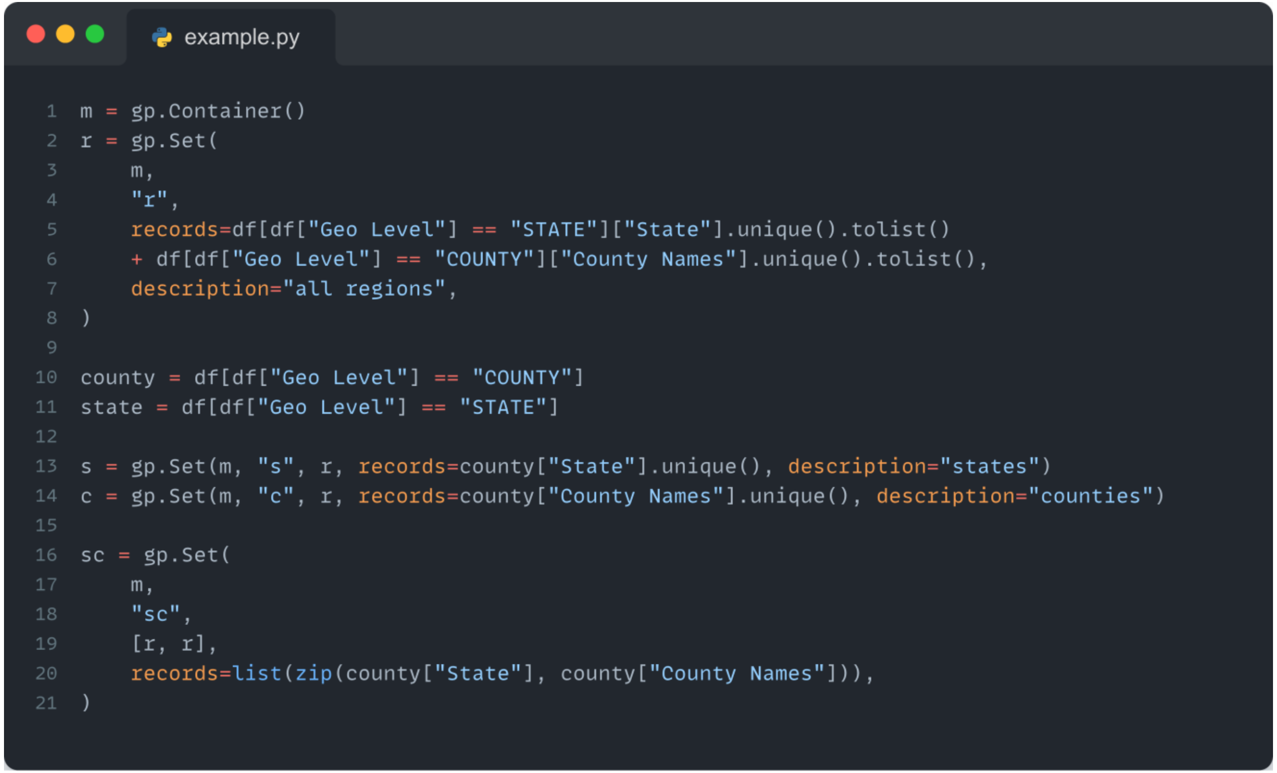


Read data

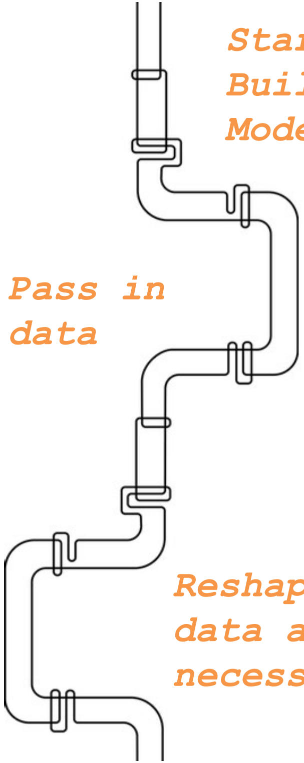
Type data

Clean data

Single Environment Pipelines



```python
m = gp.Container()
r = gp.Set(
    m,
    "r",
    records=df[df["Geo Level"] == "STATE"]["State"].unique().tolist()
    + df[df["Geo Level"] == "COUNTY"]["County Names"].unique().tolist(),
    description="all regions",
)

county = df[df["Geo Level"] == "COUNTY"]
state = df[df["Geo Level"] == "STATE"]

s = gp.Set(m, "s", r, records=county["State"].unique(), description="states")
c = gp.Set(m, "c", r, records=county["County Names"].unique(), description="counties")

sc = gp.Set(
    m,
    "sc",
    [r, r],
    records=list(zip(county["State"], county["County Names"])),
)
```

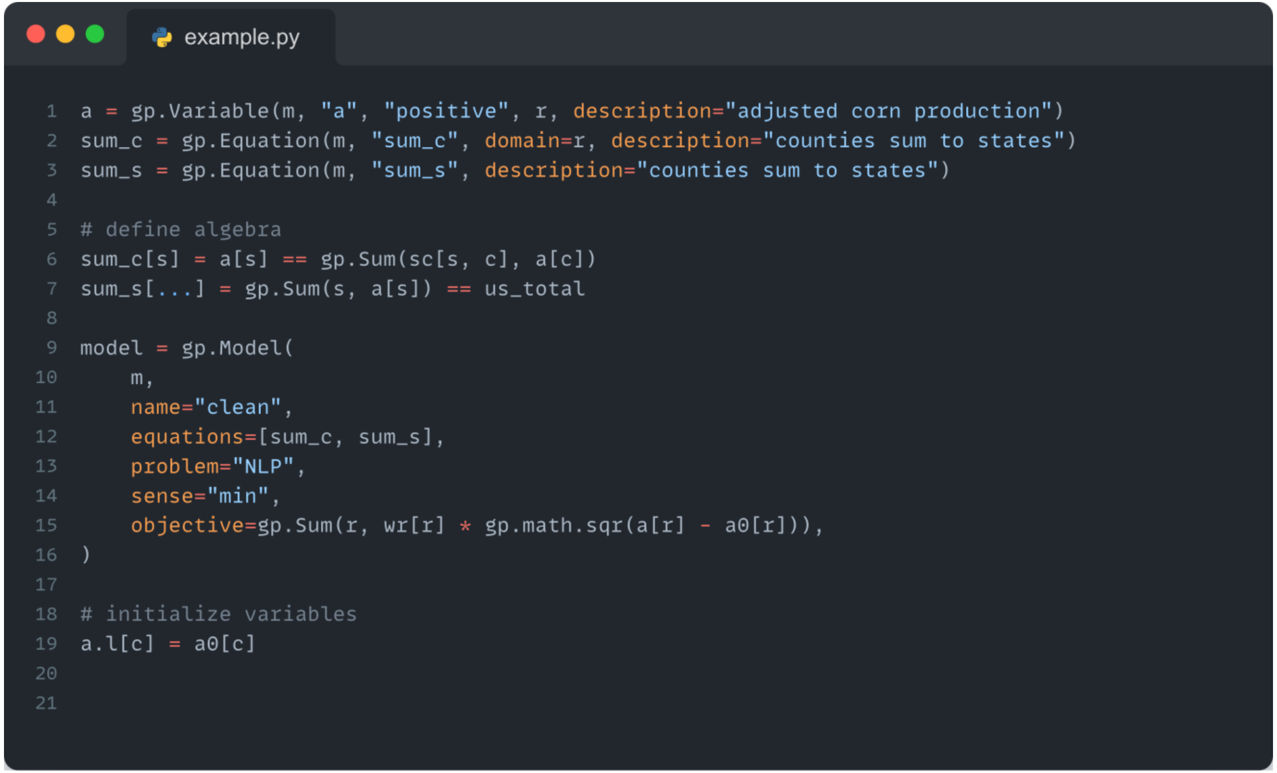


Start Building Model

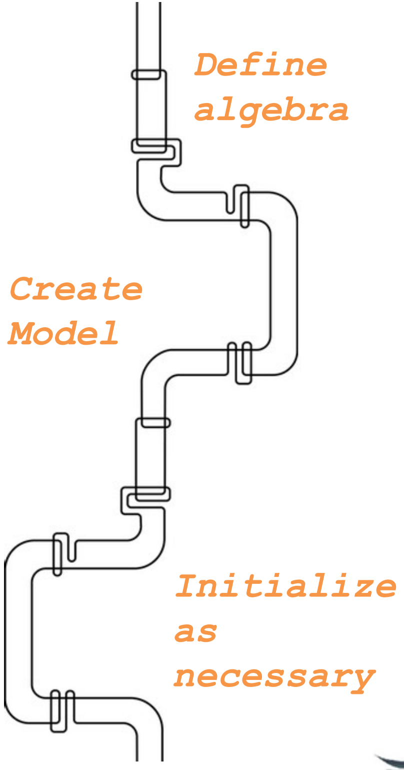
Pass in data

Reshape data as necessary

Single Environment Pipelines



```python
a = gp.Variable(m, "a", "positive", r, description="adjusted corn production")
sum_c = gp.Equation(m, "sum_c", domain=r, description="counties sum to states")
sum_s = gp.Equation(m, "sum_s", description="counties sum to states")

# define algebra
sum_c[s] = a[s] == gp.Sum(sc[s, c], a[c])
sum_s[...] = gp.Sum(s, a[s]) == us_total

model = gp.Model(
    m,
    name="clean",
    equations=[sum_c, sum_s],
    problem="NLP",
    sense="min",
    objective=gp.Sum(r, wr[r] * gp.math.sqr(a[r] - a0[r])),
)

# initialize variables
a.l[c] = a0[c]
```

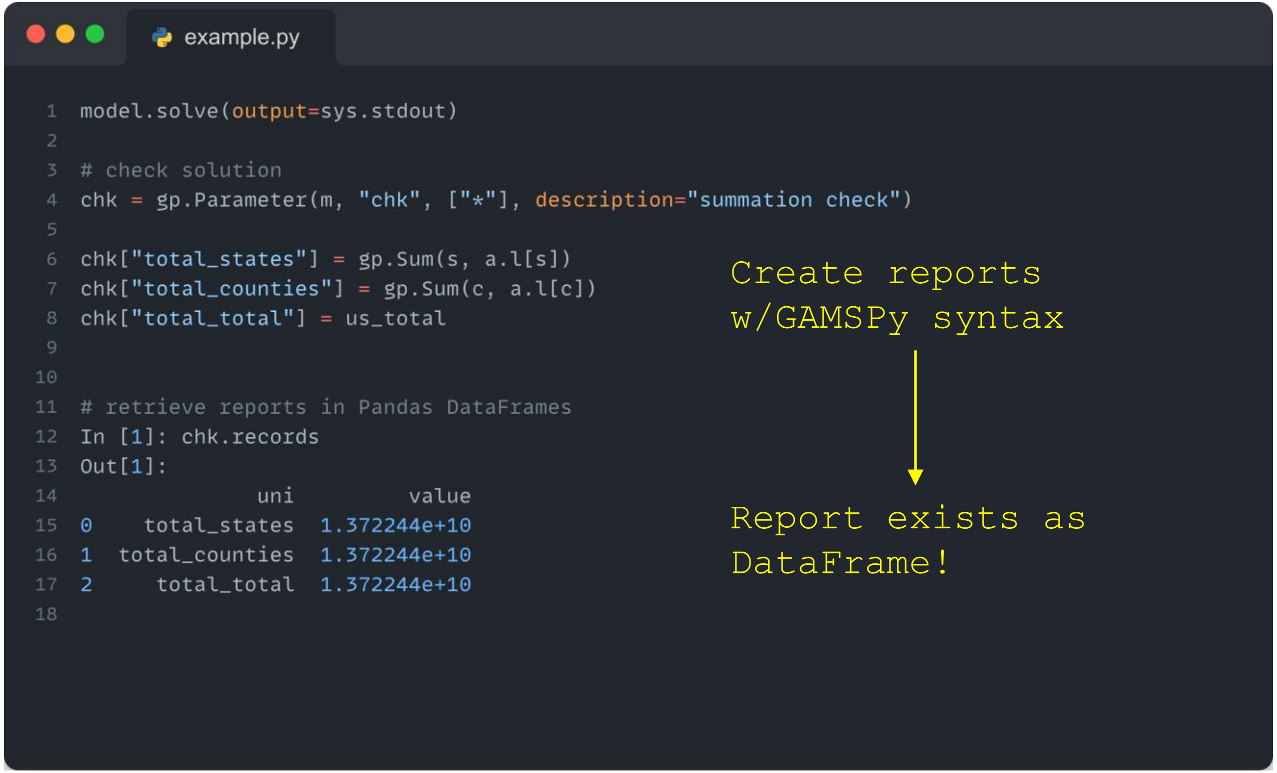


Define algebra

Create Model

Initialize as necessary

Single Environment Pipelines



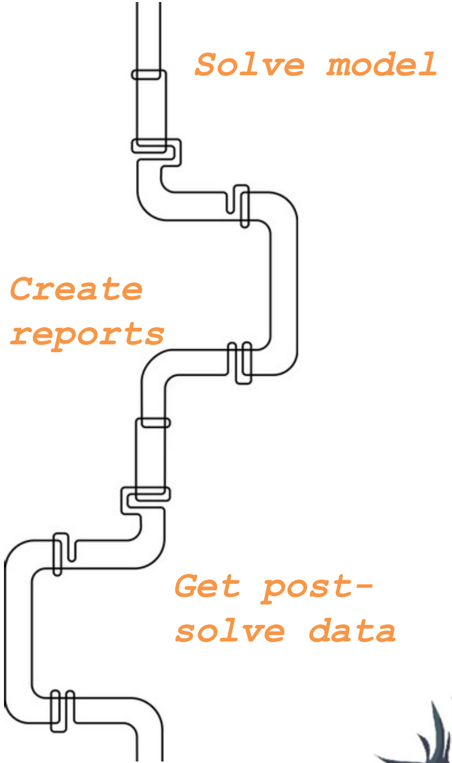
example.py

```
model.solve(output=sys.stdout)

# check solution
chk = gp.Parameter(m, "chk", ["*"], description="summation check")

chk["total_states"] = gp.Sum(s, a.l[s])
chk["total_counties"] = gp.Sum(c, a.l[c])
chk["total_total"] = us_total


# retrieve reports in Pandas DataFrames
In [1]: chk.records
Out[1]:
              uni         value
0    total_states  1.372244e+10
1  total_counties  1.372244e+10
2     total_total  1.372244e+10

```

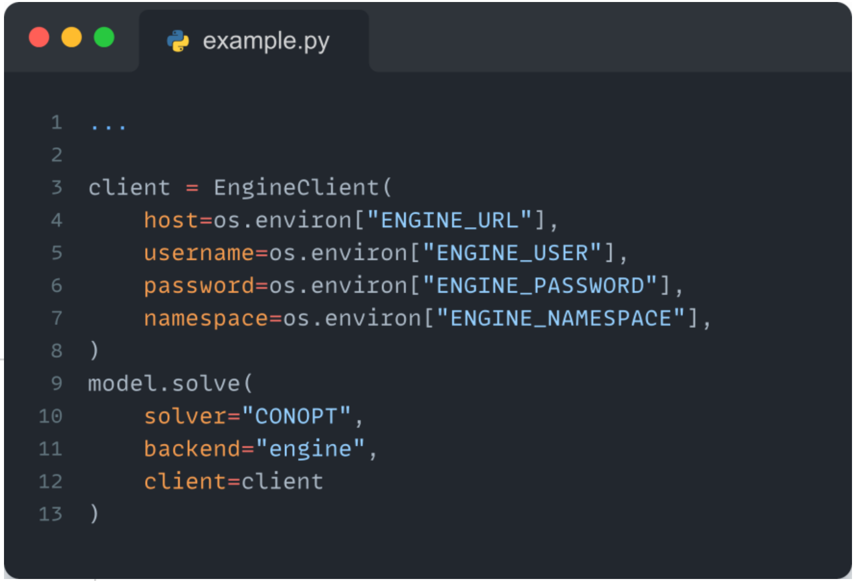
Create reports w/GAMSPy syntax

↓

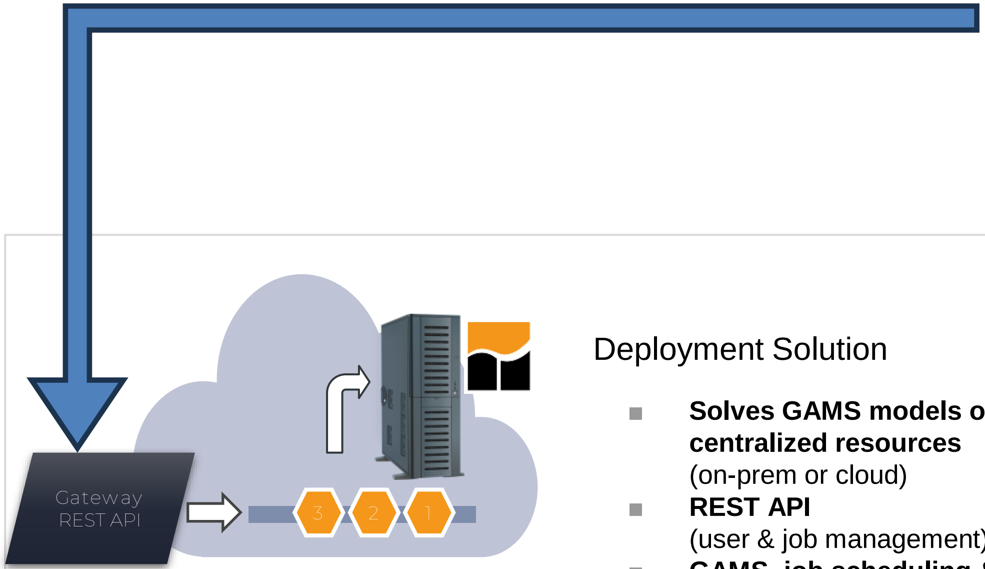
Report exists as DataFrame!



GAMSPy on cloud with GAMS Engine



```python
...

client = EngineClient(
    host=os.environ["ENGINE_URL"],
    username=os.environ["ENGINE_USER"],
    password=os.environ["ENGINE_PASSWORD"],
    namespace=os.environ["ENGINE_NAMESPACE"],
)
model.solve(
    solver="CONOPT",
    backend="engine",
    client=client
)
```



Deployment Solution

- **Solves GAMS models on centralized resources** (on-prem or cloud)
- **REST API** (user & job management)
- **GAMS job scheduling & Load balancing**

ENGINE

Summary

- Generates mathematical models (not instances) – pure representation of mathematical symbols, devoid of specific data
- GAMSPy leverages a GAMS backend to execute assignment operations, generate and solve models
- Access a broad set of state-of-the-art optimization solvers
- Unique and streamlined way to completely tasks like pre/post-processing and visualization – all in a single environment
- GAMSPy works seamlessly with GAMS MIRO, GAMS Engine, and NEOS (local machines vs. cloud/AWS machines)
- GAMSPy is fully installable with one line – `pip install gamspy`

GAMS

Contact us with your project ideas!

consulting@gams.com

support@gams.com

sales@gams.com