SUNPOWER

SunPower FOSS Projects

Mark Mikofski | September 20, 2017

About me

- @bwanamarko, @breaking_bytes, https://poquitopicante.blogspot.com
- PhD in ME from UC Berkeley
- Numerical modeling of physical phenomena for 15 years
- Lead architect of performance prediction software at SunPower
  - SunPower has been making the most efficient commercial PV solar panels for 30 years
  - Installed over 2 gigawatts of renewable energy on residential and commercial rooftops as well as utility power plants worldwide

July 28th Solar Impulse 2 became the first plane ever to circumnavigate the world using only solar energy from SunPower solar cells

I make models to simulate solar power performance at SunPower. SunPower makes and sells solar panels and solar power systems.

Agenda

- SunPower open-source software projects at https://sunpower.github.io
- Carousel Scientific Model Simulation Framework
  - structure
  - Pythagorean Theorem example
- Uncertainty Wrapper
  - Pythagorean Theorem example again
- Help Wanted!

- The goal of this talk to inspire collaboration on SunPower open source projects

SunPower Open Source Software

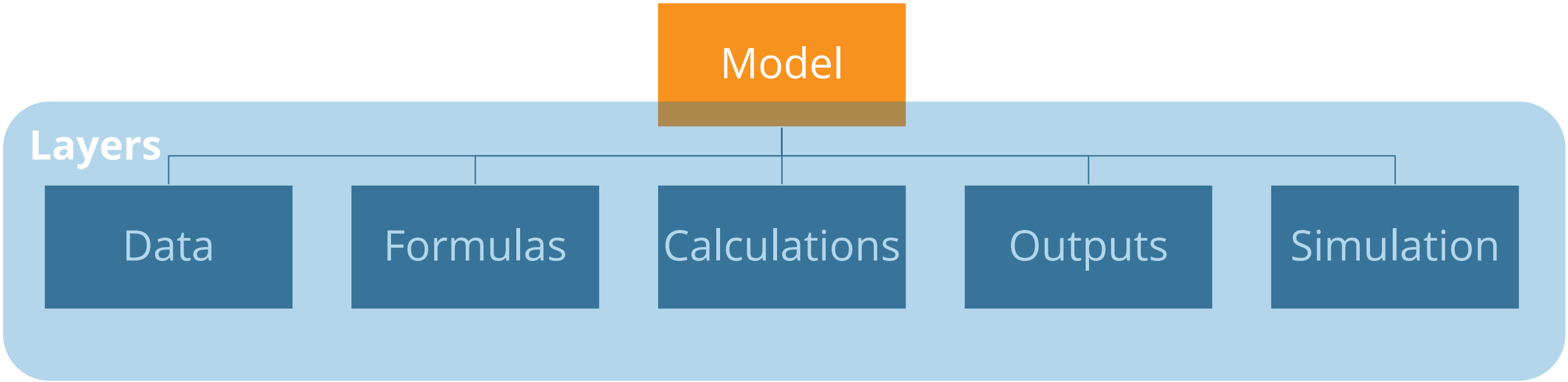
- Carousel – a framework for scientific models and simulations
- UncertaintyWrapper – error propagation using 1st order Taylor series expansions and finite central differences
- PVMismatch – toolbox for modeing solar-panel current-voltage traces with shading
- PVFree – an online database of solar power modeling parameters
- SolarUtils – Python bindings for NREL's Solar Position Calculator and SPECTRL2
- PVLIB-Python – a solar power modeling library supported by Sandia National Laboratories

- SunPower is trying to contribute to open source and reproducible science

Carousel – Scientific model simulation framework

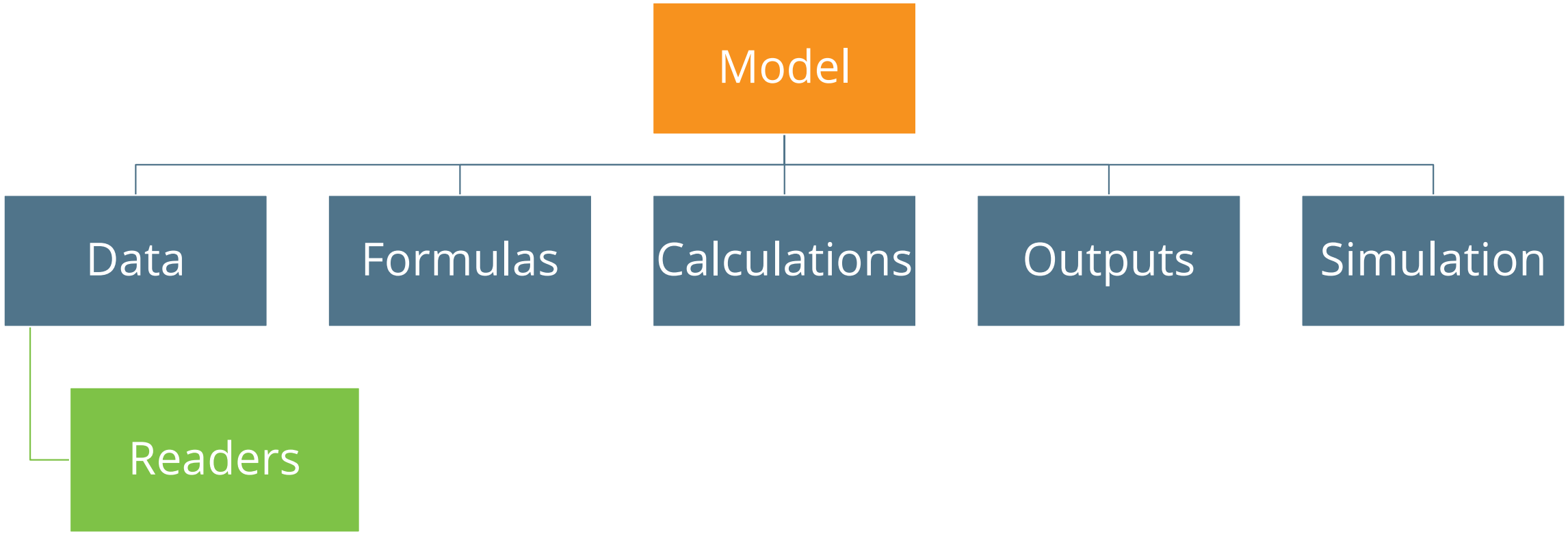
- Problem statement:
  - SunPower has mathematical models of solar power generation that are difficult to maintain because
    - Several coding paradigms and styles are mixed together, so code is difficult to understand and therefore edit
    - Boilerplate routines are repeated several times but implemented differently
    - There is no overall architecture or structure
- Proposed Solution:
  - Provide a simple framework for expressing and solving mathematical models of scientific problems
- Rationale:
  - Allow modelers to focus on implementing algorithms by separating out boilerplate routines and structure
- Use a framework for boilerplate routines, and allow modelers to focus on algorithms

Carousel – Structure



- A Carousel *Model* consists of 5 *Layers*: **Data**, **Formulas**, **Calculations**, **Outputs** and **Simulation**. Carousel can be extended by adding layers.

Carousel – Data



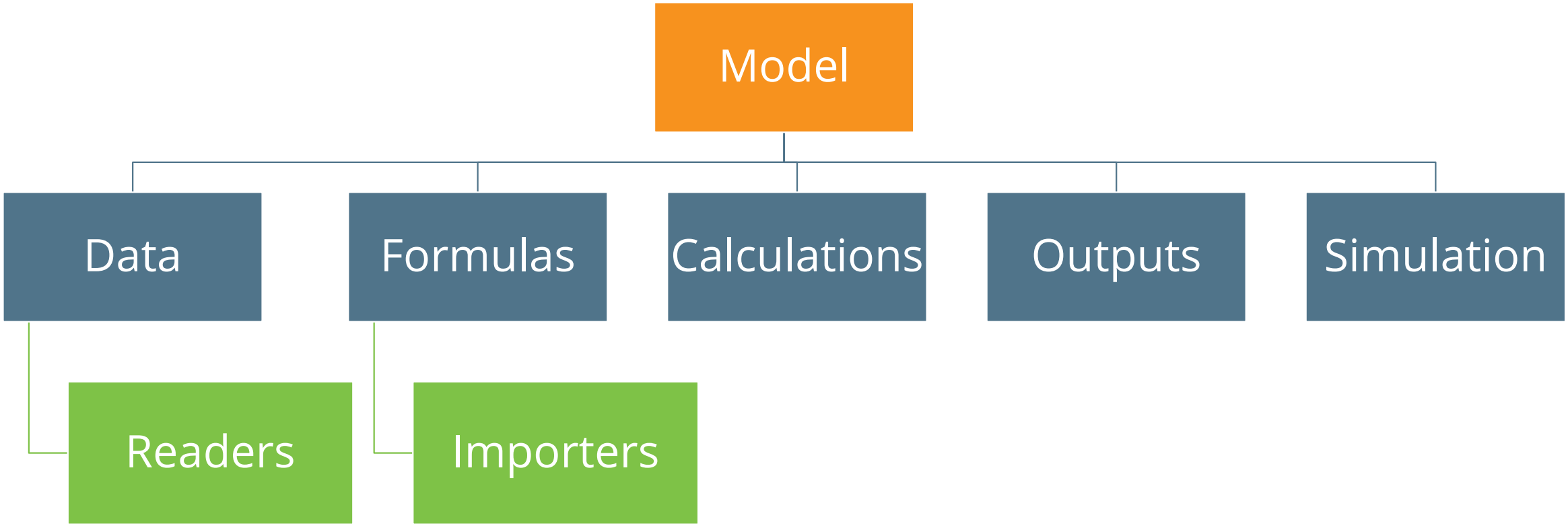
- The *Data* layer defines model inputs. **Readers** are boilerplate routines to get input from sources like CSV files, **JSON** (default), databases, API, etc.

Carousel – Data example

```
3   from carousel.core.data_sources import DataSource, DataParameter
4   from carousel.core import UREG
5
6   class PythagoreanData(DataSource):
7       adjacent_side = DataParameter(units='cm', uncertainty=1.0)
8       opposite_side = DataParameter(units='cm', uncertainty=1.0)
9
10      def __prepare_data__(self):
11          for k, v in self.parameters.iteritems():
12              self.uncertainty[k] = {k: v['uncertainty'] * UREG.percent}
13
14      class Meta:
15          data_cache_enabled = False
16          data_reader = ArgumentReader
```

- *Data* are given as class attributes defined as **`DataParameter`** which describe the data. Additional *Data* options are assigned in **`class Meta`**. Manual preparation of *Data* is executed in required **`__prepare_data__`** method, use **`pass`** to skip.

Carousel – Formulas



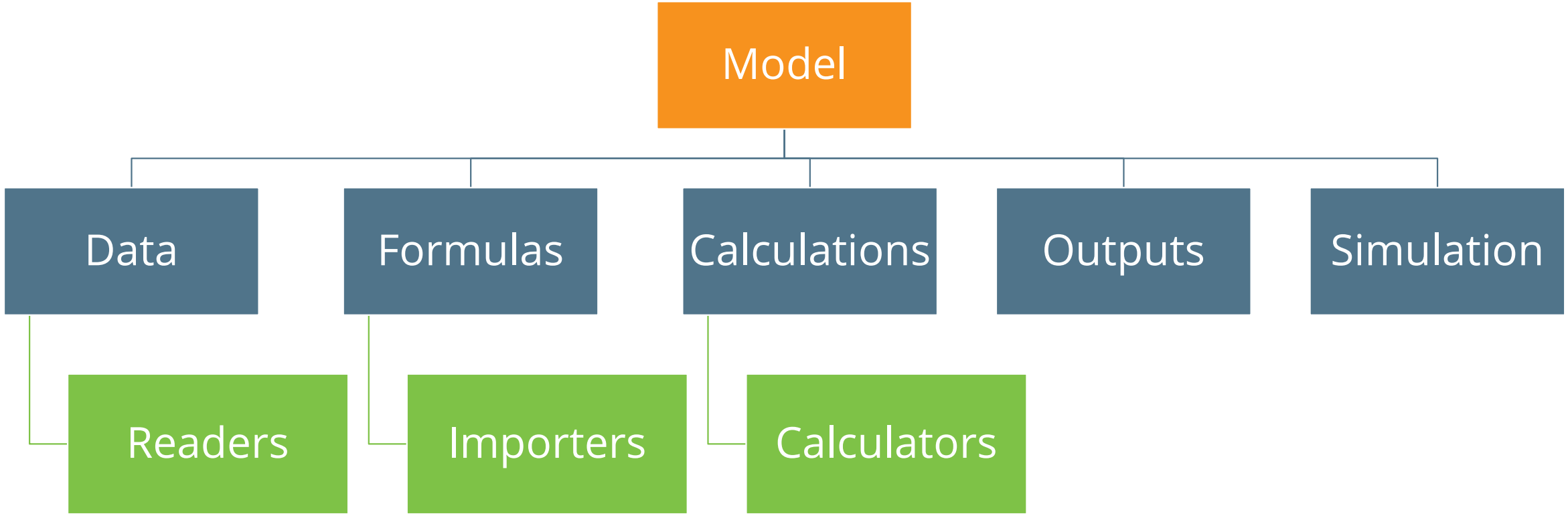
- *Formulas* are Python functions, numerical expressions or other algorithms that operate on *Data* and return *Outputs*. **Importers** are boilerplate routines that import formulas and make them callable.

Carousel – Formulas example

```
3   from carousel.core.formulas import Formula, FormulaParameter
4
5   def f_pythagorean(a, b):
6       a, b = np.atleast_1d(a), np.atleast_1d(b)
7       return np.sqrt(a * a + b * b).reshape(1, -1)
8
9   class PythagoreanFormula(Formula):
10      f_pythagorean = FormulaParameter(
11          units=[('=A', ), ('=A', '=A')],
12          isconstant=[]
13      )
14
15      class Meta:
16          module = __name__
```

- *Formulas* are class attributes defined as **`FormulaParameter`** objects. Python functions should have the same name. Additional options are in **`class Meta`**.

Carousel – Calculations



- *Calculations* are instructions for how to use *Formulas* with *Data* and *Outputs*. **Calculators** are boilerplate routines for how to execute *Calculations*.

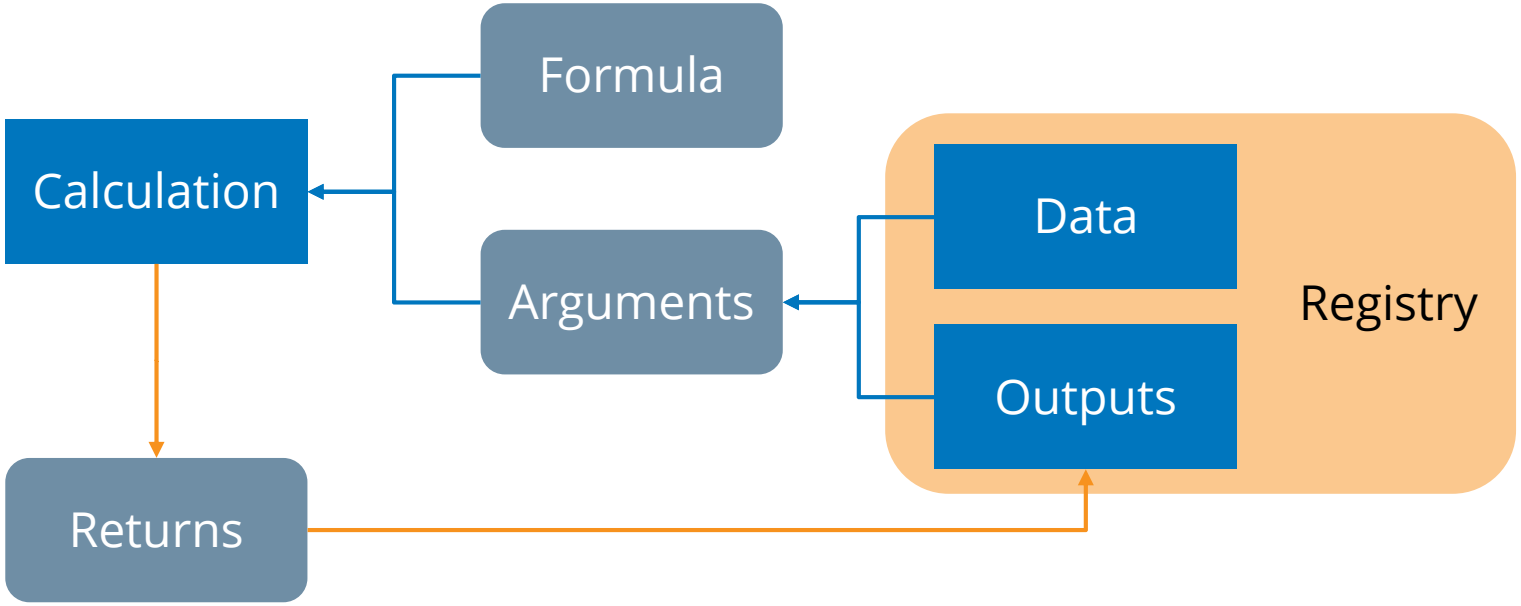
Carousel – Calculation examples

```
3  from carousel.core.calculations import Calc, CalcParameter
4
5  class PythagoreanCalc(Calc):
6      pythagorean_thm = CalcParameter(
7          formula='f_pythagorean',
8          args={'data': {'a': 'adjacent_side', 'b': 'opposite_side'}},
9          returns=['hypotenuse']
10     )
```

- *Calculations* are also class attributes defined as **`CalcParameter`** objects.

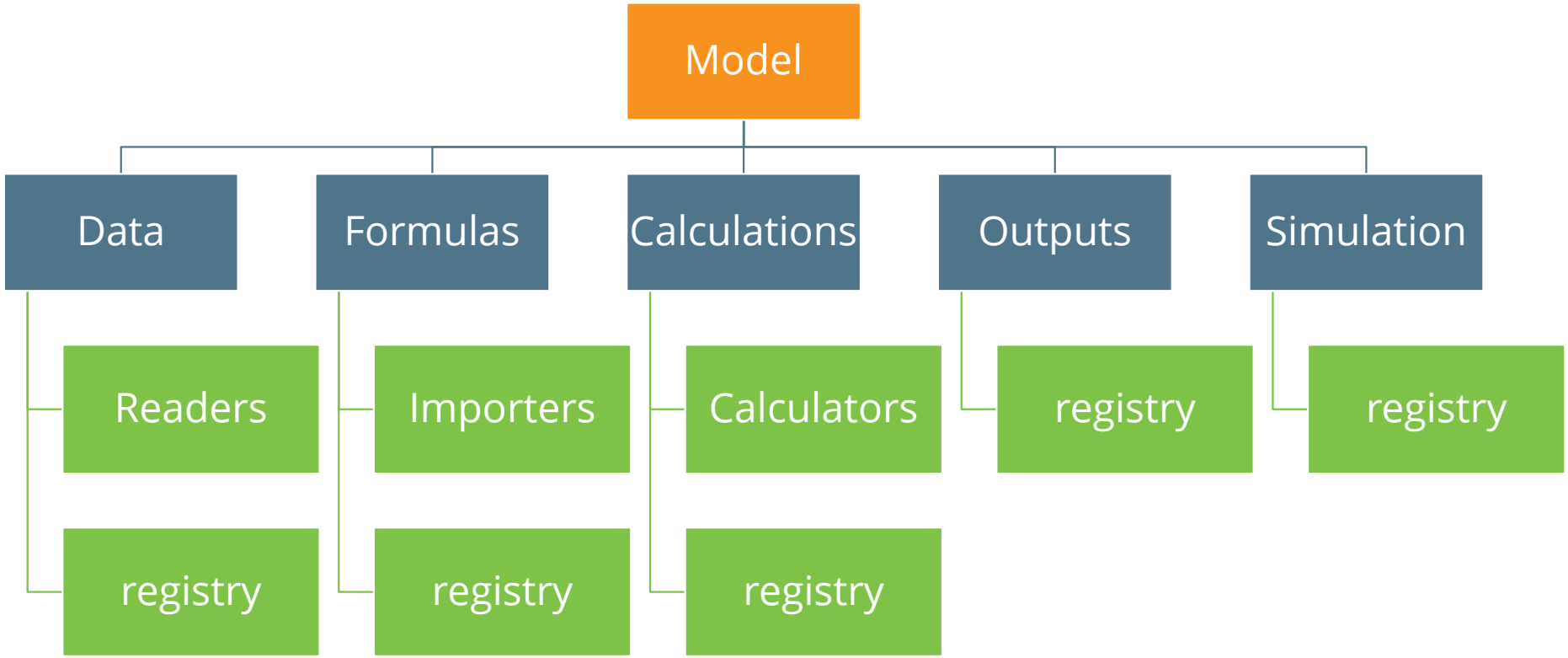
Carousel Calculations

- Calculations are instructions that map Formula arguments to Data and Outputs, and map returns to Outputs.



- Calculation simplify simulations into dynamic programs: https://en.wikipedia.org/wiki/Dynamic_programming

Carousel - Registries



- Carousel layers use a **Registry** to stores its objects. The *Output* layer stores **`OutputParameter`** objects which contain return values from *Calculations* and the *Simulation* layer stores **`SimParameter`** objects with simulation settings.

Carousel – Examples

- Clone/fork the GitHub repository: https://github.com/SunPower/Carousel
  - Try to implement a Pythagorean theorem model
  - Try to follow the Solar power system model using PVLIB-python in the Carousel tutorials
- Look at the GitHub issues
  - #110 incompatible with Pint >= 0.7.2
  - #85 treat simulation parameters the same as other layers, store `SimParameters` in Simulation registry and allow users to select which set of settings they want
- Look at the Wiki roadmap
  - Make `index_registry` a calculator method like `get_covariance`, and create `assign_registry` and `set_covariance` and other calculator output
  - Create client & server processes that disconnect the user/caller from Carousel
    - Modify simulation layer to use sub processes to run calculations concurrently according to DAG
- Contribute! What do you want to see?

UncertaintyWrapper

- Problem Statement:
  - Propagate uncertainty given a function and the covariance of the input without access inside the function itself only the ability to call functions.
  - Obtain reasonably accurate results quickly.
- Current State:
  - Use auto differentiation and methods that overload NumPy methods or perform Monte-Carlo simulations
- Solution:
  - Apply finite difference approximations and use Taylor series expansion

- UncertaintyWrapper let's you propagate uncertainty quickly without access inside functions or need to rewrite functions.

UncertaintyWrapper – Examples

- Clone/fork GitHub repo: https://github.com/SunPower/UncertaintyWrapper
  - Try to go through tutorial 3(b) in Carousel documentation
  - Try to go through examples in UncertaintyWrapper documenation
- Look at issues
  - Fix jagged arrays, currently doesn't work well if scalars and series mixed together, especially if function doesn't work with array data.
  - Can UncertaintyWrapper be merged into existing Uncertainties package? Can UncWrapper be modified to accept Uncertainties `ufloat` and other objects?
- Contribute? What do you want to do with UncertaintyWrapper? Speed it up more?

- This project is powerful, but confusing, how can it be simpler?

Thank You

Let’s change the way our world is powered.