A Class to Manage Large Ensembles and Batch Execution in Python

PyCon Canada

Andre R. Erler

November 12th, 2016
Outline

Introduction
    Science is Repetitive
    What I do

Batch Execution using an Ensemble Class
    The Ensemble Class
    A Helper Class

Argument Expansion
    Outer Product Implementation

Summary & Conclusion
Science is Repetitive

To reach conclusive results, scientific experiments usually have to be repeated many times; either to establish statistical significance, or to test a range of parameter values for optimization.

Experiments are planned and conducted in large batches or so-called *ensembles*.

Automation

It is therefore desirable to automate the most repetitive tasks, and to create tools for this purpose.
Science is Repetitive

To reach conclusive results, scientific experiments usually have to be repeated many times; either to establish statistical significance, or to test a range of parameter values for optimization.

Experiments are planned and conducted in large batches or so-called *ensembles*.

Automation

It is therefore desirable to automate the most repetitive tasks, and to create tools for this purpose.
Introduction

Ensemble Class

Argument Expansion

Rationale

Repetitive Science

What I do

Motivation

**Coupling Climate Models with Hydrologic Models**

Athabasca River watershed: groundwater depth (top) and surface water depth (bottom)

Surface Temperature in a Global and a nested Regional Climate Model

I run Climate and Hydrologic Models to study the impact of climate change on water resources and generate projections of future hydro-climate.

Andre R. Erler (A.R.Erler@gmail.com)

Large Ensembles and Batch Execution with Python
High Performance Computing

- High-resolution Climate simulations:
  - 4 days on 128 cores and 300GB of storage per model year
  - 36 ensemble members, 15 years each

- Surface-Subsurface Hydrologic Simulations:
  - 1 day on 2 cores per model year
  - also 15 years each, 100+ ensemble members
Motivation:
Batch Processing

- In Computational Sciences repetitive tasks can be automated/scripted

Python is an Ideal Scripting Language

```python
ensemble = [...]  # a list of objects ‘members’

# for loop iterating over list
tmp = []  # store results
for member in ensemble:  # iterate over list
    tmp.append(result = member.operation(*args, **kwargs))
ensemble = tmp

# list comprehension is already much shorter!
ensemble = [m.operation(*args, **kwargs) for m in ensemble]
```
**Motivation:**

**Batch Processing**

- In Computational Sciences repetitive tasks can be automated/scripted

---

**Python is an Ideal Scripting Language**

```python
ensemble = [...] # a list of objects 'members'

# for loop iterating over list
tmp = [] # store results
for member in ensemble: # iterate over list
tmp.append( result = member.operation(*args, **kwargs) )
ensemble = tmp

# list comprehension is already much shorter!
ensemble = [m.operation(*args, **kwargs) for m in ensemble]
```

---

**Boilerplate Code**

Python simplifies scripting a lot, but we still have a lot of boilerplate code! This can be simplified further.

Andre R. Erler (A.R.Erler@gmail.com)

Large Ensembles and Batch Execution with Python
Motivation: Batch Processing

- In Computational Sciences, repetitive tasks can be automated/scripted.

And Ideal Use-case Example

```python
ensemble = Ensemble(*[...])  # create Ensemble object

# apply member methods to entire ensemble
ensemble = ensemble.operation_1(*args, **kwargs)
...
ensemble = ensemble.operation_N(*args, **kwargs)

member_N = ensemble[n]  # access elements by index
member_key = ensemble[key]  # .. or by name/key
...
```

Andre R. Erler (A.R.Erler@gmail.com)
Motivation: Batch Processing

- In Computational Sciences, repetitive tasks can be automated/scripted.

The Ensemble Class

- Emulate Container Type
- Redirect method calls to ensemble members

And Ideal Use-case Example

```python
ensemble = Ensemble(*[[...]])  # create Ensemble object

# apply member methods to entire ensemble
ensemble = ensemble.operation_1(*args, **kwargs)
...
ensemble = ensemble.operation_N(*args, **kwargs)

member_N = ensemble[n]  # access elements by index
member_key = ensemble[key]  # .. or by name/key
...
```

Andre R. Erler (A.R.Erler@gmail.com)

Large Ensembles and Batch Execution with Python
The Ensemble Class

Emulating the Python Container Type:

1. Support several built-in methods, such as `__len__`, `__contains__`, `__iter__`

2. Item assignment like list or dict using `__getitem__` and `__setitem__`

Return Values

Calls to member methods return a new container or Ensemble with the results
The Ensemble Class

Emulating the Python Container Type:

1. Support several built-in methods, such as `__len__`, `__contains__`, `__iter__`

2. Item assignment like list or dict using `__getitem__` and `__setitem__`

**Return Values**

Calls to member methods return a new container or Ensemble with the results

---

**Implementation Snippet**

class Ensemble(object):
    _members = None # members

    def __getitem__(self, i):
        # get individual members
        if isinstance(i, int):
            # access like list/tuple
            return self._members[i]
        elif isinstance(i, str):
            ...

    def __iter__(self):
        # iterate over members
        mm = self._members
        return mm.__iter__()
The Ensemble Class

Implementation of Method Redirection:

1. Redirect calls to member methods/attributes by overloading `__getattr__`

2. Execute call on all Ensemble members

3. Return a new container or Ensemble with results

Ensemble Wrapper

Methods require helper Class `EnsWrap` to apply arguments

```
class Ensemble(object):
    _members = None  # members
    ...

    def __getattr__(self, attr):
        # check if callable
        mem0 = self._members[0]
        # assuming homogeneity...
        f = getattr(mem0,attr)
        if callable(f):
            # return Ensemble Wrapper
            v = EnsWrap(self,attr)
        else:
            # just return values
            v = [getattr(m,attr) for m in self._members]
        return v
```

Andre R. Erler (A.R.Erler@gmail.com)
The Ensemble Class

Implementation of Method Redirection:

1. Redirect calls to member methods/attributes by overloading `__getattr__`
2. Execute call on all Ensemble members
3. Return a new container or Ensemble with results

Ensemble Wrapper

Methods require helper Class EnsWrap to apply arguments

Implementation Snippet

```python
class Ensemble(object):
    _members = None # members

    def __getattr__(self, attr):
        # check if callable
        mem0 = self._members[0]
        # assuming homogeneity...
        f = getattr(mem0,attr)
        if callable(f):
            # return Ensemble Wrapper
            v = EnsWrap(self,attr)
        else:
            # just return values
            v = [getattr(m,attr) for m in self._members]
        return v
```
A Helper Class

Implementation of the Ensemble Wrapper:

1. Initialize with ensemble members and the called attribute/method

2. Use `__call__` method to execute member method with arguments

Parallelization

Simple parallelization using `multiprocessing.Pool`'s `apply_async` can be applied.
A Helper Class

Implementation of the Ensemble Wrapper:

1. Initialize with ensemble members and the called attribute/method

2. Use __call__ method to execute member method with arguments

Parallelization

Simple parallelization using multiprocessing.Pool's apply_async can be applied
How can we use Ensembles with Argument Lists

A Trivial Case

```python
# this defeats the purpose
members = [member.operation(arg1=arg) for arg in arg_list]
Ensemble(*members) # initialize new ensemble

# a better solution: pass list directly
ensemble.operation(arg1=arg_list, inner_list=['arg1'])
```

Argument lists can easily be implemented in the `__call__` method of the ensemble wrapper EnsWrap by creating a list of arguments for each member

```python
# construct argument list
args_list = expandArgList(**kwargs)
# loop over lists
ens = self._ensemble
for m, args in zip(ens, args_list):
    f = getattr(m, self.attr)
    # execute member method with args
    new.append(f(**args))
```
How can we use Ensembles with Argument Lists

A More Complex Case: the Outer Product List

```python
# again, this defeats the purpose
arg_list = []
for arg1 in arg_list1:  # construct arg_list from two lists
    for arg2 in arg_list2:  # i.e. all possible combinations
        arg_list.append(dict(arg1=arg1, arg2=arg2))
# apply list to ensemble
ensemble.operation(arg1=arg_list, inner_list=[‘arg1’])

# a better solution is to expand the lists internally
ensemble.operation(arg1=arg_list1, arg2=arg_list2,
    outer_list=[‘arg1’, ‘arg2’])
```

The **Outer Product** expansion of multiple argument lists creates argument lists with all possible combinations of arguments. **Inner Product** expansion works like Python’s `zip` function.
Argument Expansion via Outer Product

Recursive Implementation of Outer Product:

1. Separate expansion arguments from others
2. Recursively expand argument list
3. Generate argument set for each ensemble member

Decorator Class

Argument Expansion is most useful as a Decorator class

Implementation of Recursion

```python
def expandArgsList(args_list, exp_args, kwargs):
    # check recursion condition
    if len(exp_args) > 0:
        # expand arguments
        now_arg = exp_args.pop(0)
        new_list = []  # new arg list
        for narg in kwargs[now_arg]:
            for arg_list in args_list:
                arg_list.append(narg)
                new_list.append(arg_list)
        # next recursion level
        args_list = expandArgsList(new_list, exp_args, kwargs)
    ...
    # terminate: return arg lists
    return args_list
```

Andre R. Erler (A.R.Erler@gmail.com)

Large Ensembles and Batch Execution with Python
Argument Expansion via Outer Product

Recursive Implementation of **Outer Product**:

1. Separate expansion arguments from others
2. Recursively expand argument list
3. Generate argument set for each ensemble member

**Decorator Class**

Argument Expansion is most useful as a Decorator class

```python
def expandArgsList(args_list, exp_args, kwargs):
    # check recursion condition
    if len(exp_args) > 0:
        # expand arguments
        now_arg = exp_args.pop(0)
        new_list = []  # new arg list
        for narg in kwargs[now_arg]:
            for arg_list in args_list:
                arg_list.append(narg)
                new_list.append(arg_list)
        # next recursion level
        args_list = expandArgsList(new_list, exp_args, kwargs)
    ...
    # terminate: return arg lists
    return args_list
```

Andre R. Erler (A.R.Erler@gmail.com)

Large Ensembles and Batch Execution with Python
Argument Expansion via Outer Product

Recursive Implementation of Outer Product:

1. Separate expansion arguments from others
2. Recursively expand argument list
3. Generate argument set for each ensemble member

Decorator Class

Argument Expansion is most useful as a Decorator class

Implementation of Recursion

```python
def expandArgsList(args_list, exp_args, kwargs):
    # check recursion condition
    if len(exp_args) > 0:
        # expand arguments
        now_arg = exp_args.pop(0)
        new_list = []  # new arg list
        for narg in kwargs[now_arg]:
            for arg_list in args_list:
                arg_list.append(narg)
                new_list.append(arg_list)
        # next recursion level
        args_list = expandArgsList(new_list, exp_args, kwargs)
    ...
    # terminate: return arg lists
    return args_list
```
Summary & Conclusion

The Ensemble Class

- Functions like a **container type** and redirects calls to (parallelized) **member methods**

Argument Expansion

- Systematic expansion of argument lists from **inner** or **outer product** (with decorator)

Sprint Project: Publish Ensemble Class

Create a stand-alone module with the Ensemble class and the argument expansion code for others to use, and add support for array-like item access/assignment
Thank You! ~ Questions?