The "datatype" Concept and Functionality

Simon Albright

BE-RF-BR

4 - December - 2020

Contents

- 28th February 2020: `https://indico.cern.ch/event/886262/`
- 11th March 2020: `https://indico.cern.ch/event/895758/`
- 9th April 2020: `https://indico.cern.ch/event/907826/`
- 17th April 2020: `https://indico.cern.ch/event/909979/`

A datatype object:

1. Is a *numpy* array (*np.ndarray* is used as a base class)
2. Is initialised in a flexible, but strictly defined way
3. Self-identifies by storing important information
4. Self-interpolates
5. Knows how to be transformed

Numpy array

```
1 class datatype_base(np.ndarray):
2     def __new__(cls, input_array, [other inputs]):
3         obj = np.asarray(input_array).view(cls)
4         obj.use_input(other_inputs)
5         return obj
```

- All *numpy* functionality is preserved (slicing, vectorisation, etc)
- Where beneficial, *numpy* features can be overwritten
- Additional functionality can be added

Initialisation (*i*)

```
class datatype_derived(datatype_base):
    def __new__(cls, *args, [other inputs]):
        #do stuff
        return obj
```

- **args* allows an arbitrary number of inputs
- Using **args* instead of a *list* or *tuple* (or other) removes ambiguity with:
  - ***[list, list]***: 2D array of [Time, Value] or [Value, Value]?
  - ***list***: List of single values or value specified on turn numbers?

Initialisation (*ii*)

```
class datatype_derived(datatype_base):
    def __new__(cls, *args, [other_inputs]):
        for a in args:
            if isinstance(a, numbers.Number):
                #Treat as fixed value
            if a.shape[0] == 1:
                #Treat as turn-by-turn [values]
            if a.shape[0] == 2:
                #Treat as [time, value]
```

- Each element of **args* can be single valued, or have shape *(n,)* or *(2, n)*
  - Single valued is interpreted as constant through time
  - Shape *(n,)* is interpreted as turn-by-turn
  - Shape *(2, n)* is interpreted as [time, value]
- Removes need for content to be inferred from type
- Single values can be mixed with anything, turn-by-turn cannot mix with [time, value]

Initialisation (*iii*)

At initialisation, the inputs get treated to produce a regular array of defined shape:

- All single valued: 1D array with as many inputs as elements
- 1D (+ single valued): 2D array with shape (n_elements, n_turns)
  - Single valued elements duplicated on all turns
  - Turn-by-turn data with mismatched numbers of turns not allowed
- 2D (+ single valued): 3D array with shape (n_elements, 2, n_points)
  - Single valued elements duplicated to all time points
  - If allowed, unique times extracted and all inputs interpolated to common time base
  - The type of interpolation can be specified at initialisation

Self-identification

```
class bunch_length(datatype_base):
    def __new__(cls, *args, units='s', length_type='full',
                [other_inputs]):
        #Do stuff
        return obj
```

- Class name and type specifies contents, e.g.:
  - bunch_length
  - synchronous_phase
  - momentum_program
  - voltage_function
- Important details can be specified, e.g. for bunch length:
  - units: s, ns, degrees, ...
  - type: 4sigma, RMS, FWHM, ...

Self-interpolation

- *numpy reshape* function overwritten to take:
  - Number of elements
  - Times
  - Turn numbers
- Times can be used with single valued or time based arrays
- Turn numbers can be used with single valued or turn based arrays
- For time based, interpolation method is specified at initialisation and stored internally

```
1 voltages = voltage_program([time1, voltage1],
2                            [time2, voltage2], ...)
3 interp_voltages = voltages.reshape(use_time = [0, 1, 2])
```

Transformation (*i*)

- Some functions are expected to be used to calculate others, e.g.:
  - transformation between momentum, kinetic energy, total energy, bending field
  - frequency offset to phase offset
- Because the array self-identifies, transformations can be done with definite knowledge of source

```
BField = bending_field_program(*args)
momentum = BField.to_momentum(rest_mass = m_p,
                              bending_radius = rho,
                              charge = 1)
kin_energy = momentum.to_kin_energy(rest_mass = m_p,
                                    bending_radius = rho,
                                    charge = 1)
```

Transformation (*ii*)

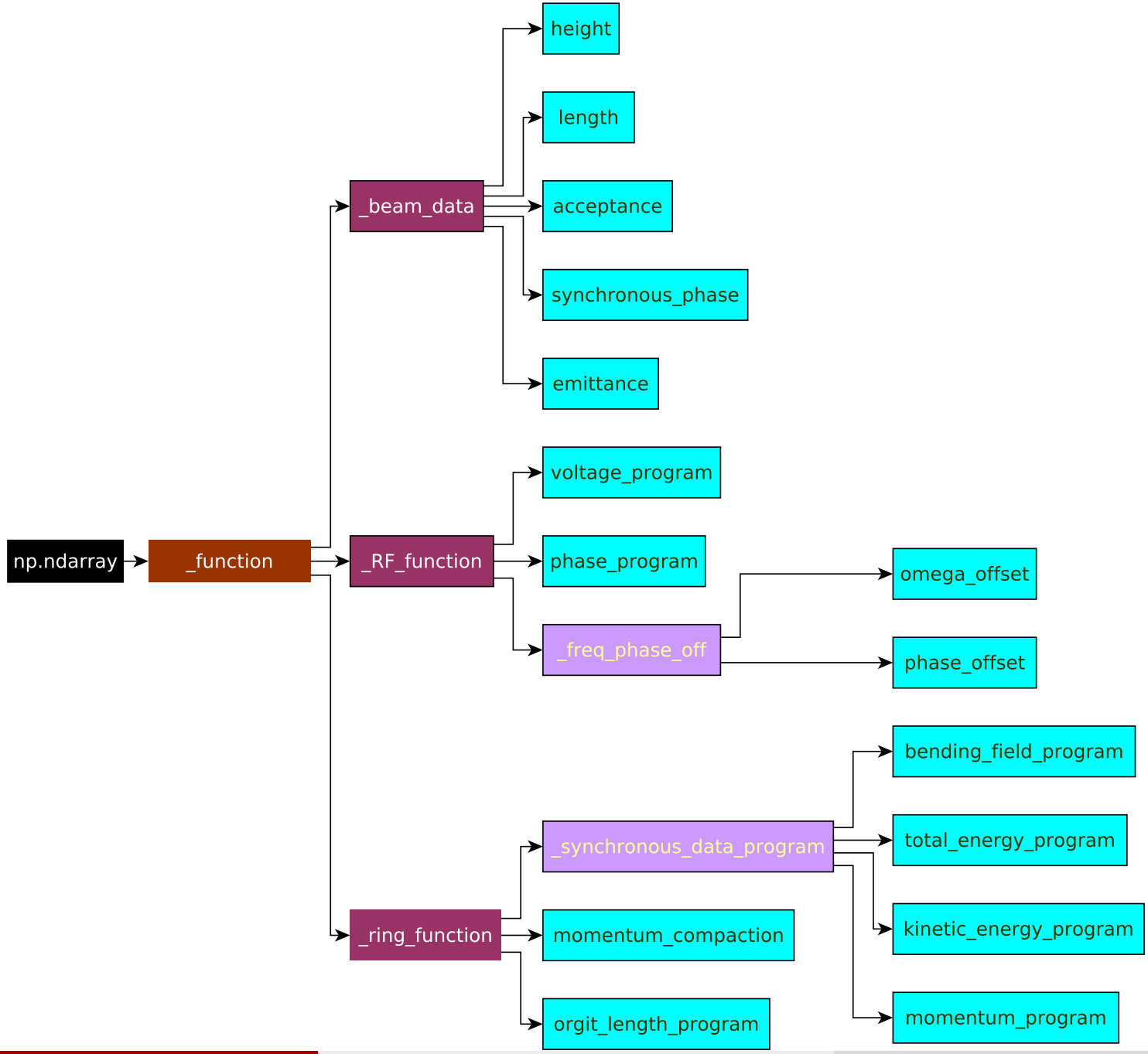
- Preprocessing is built-in to *synchronous data* type arrays
- Self identification removes need to know contents

```
data = kinetic_energy_program([time, value], ...)
data.convert(rest_mass, bending_radius, charge)
interpolated = data.preprocess(...)

data = momentum_program([time, value], ...)
data.convert(rest_mass, bending_radius, charge)
interpolated = data.preprocess(...)

data = bending_field_program([time, value], ...)
data.convert(rest_mass, bending_radius, charge)
interpolated = data.preprocess(...)
```

- **_core.py**: Base class, redefinition of *magic* functions, various support functions
- **ring_programs.py**: momentum compaction, kinetic energy, momentum, etc
- **rf_programs.py**: voltage program, phase program, frequency offset, etc
- **beam_data.py**: acceptance, length, synchronous phase, etc
- **functions.py**: Redeclaration of useful numpy functions, e.g. *vstack*


height
length
_beam_data
acceptance
synchronous_phase
emittance
voltage_program
np.ndarray
_function
_RF_function
phase_program
omega_offset
_freq_phase_off
phase_offset
bending_field_program
_synchronous_data_program
total_energy_program
kinetic_energy_program
_ring_function
momentum_compaction
momentum_program
orgit_length_program

- Save/load functions
  - .npz format for python only cases
  - .json for human readability and transferability
- More preprocess functionality (e.g. multi-section)
- Expand support functions
- *Features available on request*