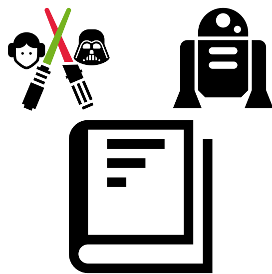
`galactic` developer guide

The Galactic Organization <contact@thegalactic.org>



0.4.0

Contents

Preamble **2**

1 GALACTIC platform **2**

2 Data readers **4**

3 Characteristics **4**

4 Descriptions **5**

5 Strategies **7**
5.1 Basic strategies 8
5.2 Meta strategies 8

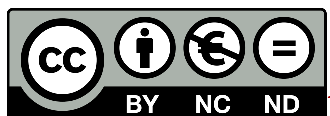
6 Measures **10**

References **11**

List of Figures

1 the NextPriorityConcept algorithm process 3
2 Characteristic plugin 5
3 Description plugin 6
4 Basic strategy plugin 9
5 Meta strategy and measure plugins 10

Preamble



[1]

1 GALACTIC platform

GALACTIC is a development platform for a generic implementation of the NextPriorityConcept algorithm (Demko et al. 2020) allowing easy integration of new plugins for characteristics, descriptions, strategies and measures useful for meta-strategies.

The **GALACTIC** eco-system is organized with:

[1] © 2018-2022 the Galactic Organization. This document is licensed under CC-by-nc-nd (https://creativecommons.org/licenses/by-nc-nd/4.0/deed.en)

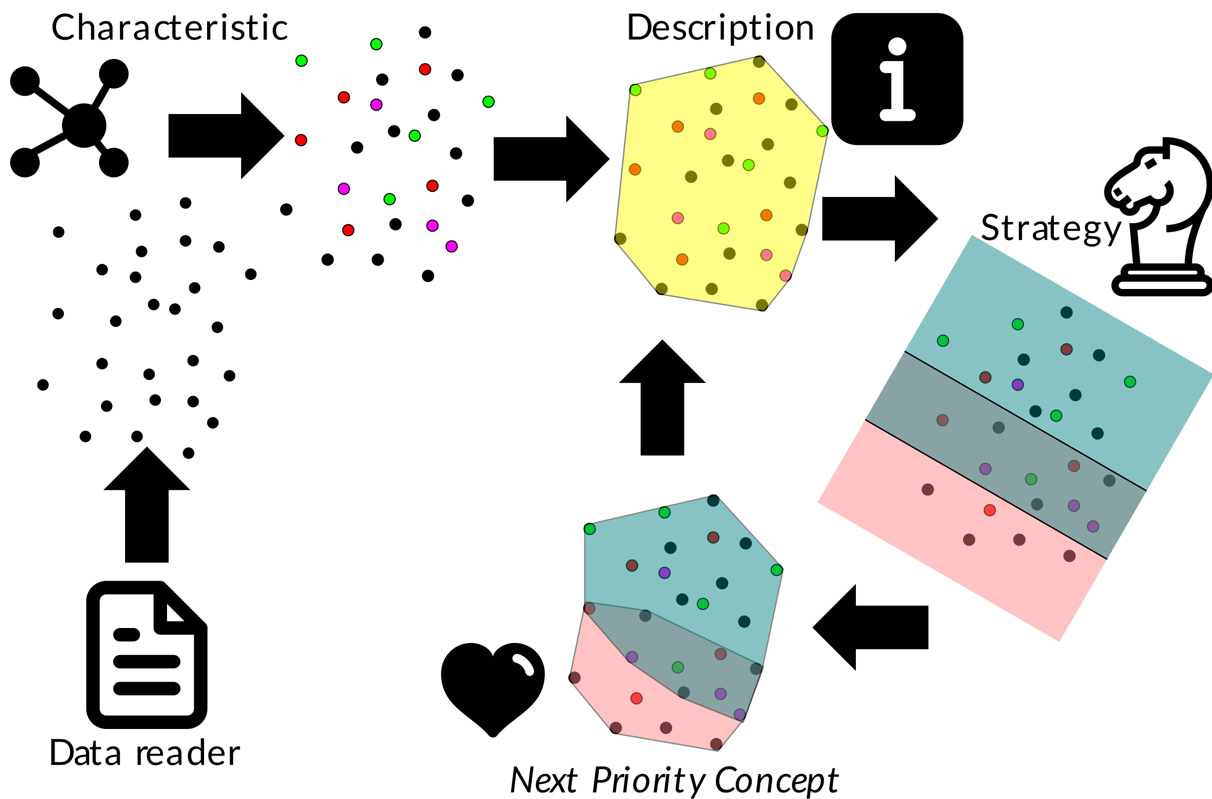


Figure 1: the NextPriorityConcept algorithm process

- A **core** which implements the NextPriorityConcept algorithm and a lot of tool for visualizing lattices and reduced contexts in python notebooks;
- A set of **characteristic plugins** defining new types of data;
- A set of **description plugins** defining new types of descriptions and their associated predicates;
- A set of **strategy plugins** defining new types of strategies for a given characteristic;
- A set of **measure plugins** usefull for the filter meta-strategies;
- A set of **data reader plugins** allowing **GALACTIC** to read any type of data file;
- A set of **applications** using the core library and the different plugins;
- A set of **localization plugins** for translating the different applications.

Each plugin must register with the core library by declaring an entry point in the configuration file of the `setuptools`[2] (`setup.py`) named `py_galactic_extension`.

```
entry_points={
    "py_galactic_extension": [
        "my_plugin = my_plugin:get_classes"
    ]
},
```

The declared function (`get_classes` in the example) will inform the library that a new extension is available.

[2] https://pypi.org/project/setuptools/

The construction of the lattice is carried out as shown in Fig. 1: the data are read from a file; characteristics are extracted; a description is produced for each concept; strategies generate selectors for the exploration of potential new concepts and the NextPriorityConcept algorithm selects the predecessors and maintains the lattice structure.

2 Data readers

Data readers are plugins of the *py-galactic-core* engine that allow to read new data file formats.

A data reader plugin must declare a class inheriting from the `galactic.concepts.DataReader` class and implementing the `read` method and defining the `extensions` property.

- `read(cls, data_file: TextIO): Iterable[Any]` that reads a file and produces an iterable of objects. The `read` method is responsible for reading an already opened file and must return the data read either as a dictionary (if the individuals are named) or as a list.
- `extensions(cls): Iterator[str]` that produces the list of file extensions supported by this plugin. The `extensions` property must return an iterator over all the file extensions supported by the plugin.

```
import io
from typing import Union, Iterable, Mapping

from galactic.concepts import DataReader

class MyDataReader(DataReader):
    @classmethod
    def read(cls, data_file: io.TextIOBase) -> Union[Iterable, Mapping]:
        # must return the data read from data_file

    @property
    def extensions(self):
        # must return an iterator over the supported file extensions
        # here: .my
        return iter([".my"])

def get_reader():
    return MyDataReader()
```

3 Characteristics

Each concept is composed of a subset of objects together with a set of predicates describing them, each predicates being specific to one type of characteristic. Such generic use of predicates makes it possible to consider heterogeneous data as input, i.e. digital, discrete or more complex data.

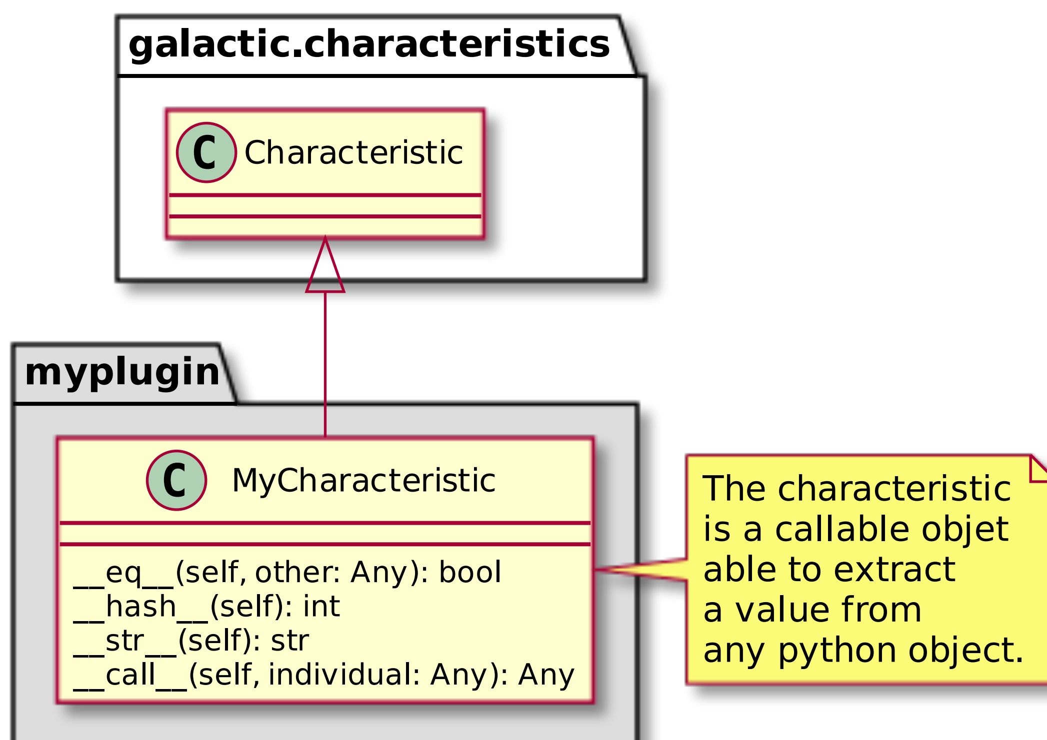


Figure 2: Characteristic plugin

A characteristic plugin (cf FIG. 2) is responsible of extracting a value from a python object. It must implement the `__call__` magic method which will be applied to each individual of the population. This method should return the characteristic of the individual. It's also reasonable to implement the other magic methods `__eq__`, `__hash__`, `__str__`.

Characteristic plugins are plugins of the *py-galactic-core* engine that allow to define new characteristics on individuals.

An characteristic plugin should declare a class inheriting from the

`galactic.characteristics.Characteristic`

class or preferably from one of its subclass and implementing the `__call__` method.

```
from galactic.characteristic import Characteristic

class MyCharacteristic(Characteristic):
    def __call__(self, individual=None):
        # Return something
```

4 Descriptions

The algorithm introduces the notion of *description* δ as an application to provide predicates describing a set of objects A according to their characteristics, that corresponds to a concept $(A, \delta(A))$. At each

iteration, predicates describing the objects A of the current concept are computed "on the fly" by a specific treatment for each type of characteristics, depending on whether it's digital, discrete or more complex, and the final description δ is the union of these predicates. In order to obtain a lattice, the description must verify $\delta(A) \sqsubseteq \delta(A')$ for $A' \subseteq A$. Let us notice that this property is verified by the "generalized convex hull" of a set of objects (the intersection of two convex hull is a convex hull), therefore predicates describing the borders of a convex hull can be used as a description.

A description plugin (cf FIG. 3) will be used to describe a collection of individuals using a set of predicates. It usually defines a new predicate class and a new description class. The description class is responsible of calculating a set of descriptors representing the convex hull of a collection of individuals. These descriptors must be predicates that describe half-spaces on the set of individuals. The convex hull is represented by the intersection of the set of descriptors.

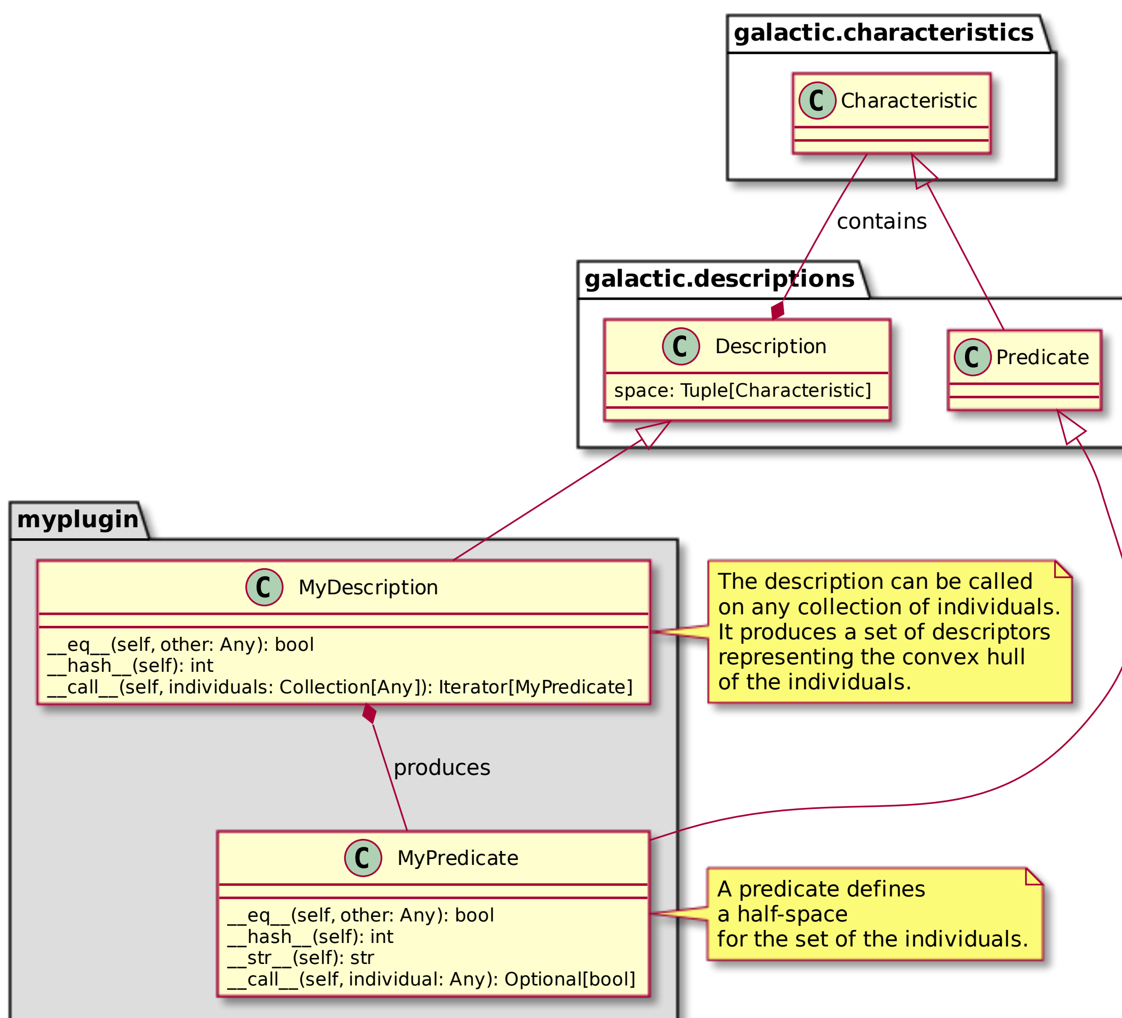


Figure 3: Description plugin

Description plugins are plugins of the *py-galactic-core* engine that allow to define new predicates and new description spaces on individuals.

A description plugin should declare a class inheriting from the

`galactic.descriptions.Predicate`

class or preferably from one of its subclass and implementing the `__call__` method. This method must return `True` or `False`.

```
from galactic.descriptions import Predicate

class MyPredicate(Predicate):
    def __call__(self, individual=None):
        # Return True or False
```

Usually, a description plugin declares a class inheriting from the

`galactic.descriptions.Description`

class and implementing the `__call__` method. This method must yield a set of minimal predicates describing a set of individuals.

```
class MyDescription(Description):
    def __call__(
        self,
        individuals: Iterable = None
    ) -> Iterator[Predicate]:
        # yield instances of MyPredicate
```

5 Strategies

The algorithm also introduces the notion of *strategy* σ to provide selectors generating the predecessors of a concept $(A, \delta(A))$. The selectors propose a way to refine or cut the description $\delta(A)$. The purpose of a strategy plugin is then to produce a set of selectors restricting the set of individuals in order to obtain new potential concepts containing fewer individuals. The NextPriorityConcept algorithm will select among the set of selectors produced by the strategies those that will give rise to effective concepts (i.e. with a reduction of individuals), while maintaining the constraints retaining the lattice property.

Several strategies are possible to generate predecessors of a concept, going from the naive strategy classicaly used in FCA that consider all the possible selectors to generate a maximal number of predecessors, thus a huge lattice. Let us observe that selectors are only used for the predecessors generation, they are not kept neither in the description or in the final set of predicates. Therefore, choosing or testing several strategies at each iteration in a user driven pattern discovery approach would be interesting. It is also possible to introduce a filter (or meta-strategy) on the selectors.

Strategies are plugins of the *py-galactic-core* engine that allow to create new predicates applicable to individuals.

There are two kinds of strategies:

- basic strategy
- meta strategy

5.1 Basic strategies

A basic strategy plugin proposes selectors to the NextPriorityConcept algorithm. These selectors make it possible to restrict the set of individuals. A basic strategy must be initialized with a description and must implement the `selectors` method (cf Fig. 4).

A basic strategy plugin must declare a class inheriting from the

`galactic.strategies.BasicStrategy`

class and implementing the `selectors` method. This method must produce an iterator over predicates defined by a description plugin.

```
from typing import Iterable

from my_characteristic_plugin import MyCharacteristic
from my_description_plugin import MyDescription, MyPredicate
from galactic.strategies import BasicStrategy
from galactic.concepts import Concept


class MyBasicStrategy(BasicStrategy):

    def __init__(self, description: MyDescription):
        # initialize the strategy

    def selectors(self, concept: Concept):
        # yield instances of MyPredicate

def get_classes():
    return {"strategy.myplugin.MyBasicStrategy": MyBasicStrategy}
```

5.2 Meta strategies

The core of the **GALACTIC** library defines two meta-strategies that act as filters for other strategies:

- `LimitFilter` which selects predecessors whose measure is above/below a threshold;
- `SelectionFilter` which selects the best/worst predecessors relatively to a measure.

They are initialized with a set of strategies and with a measure. A measure is a class for measuring a predicate on a concept. It must implement the `__call__` magic method with a concept and a predicate as parameters (cf Fig. 5).

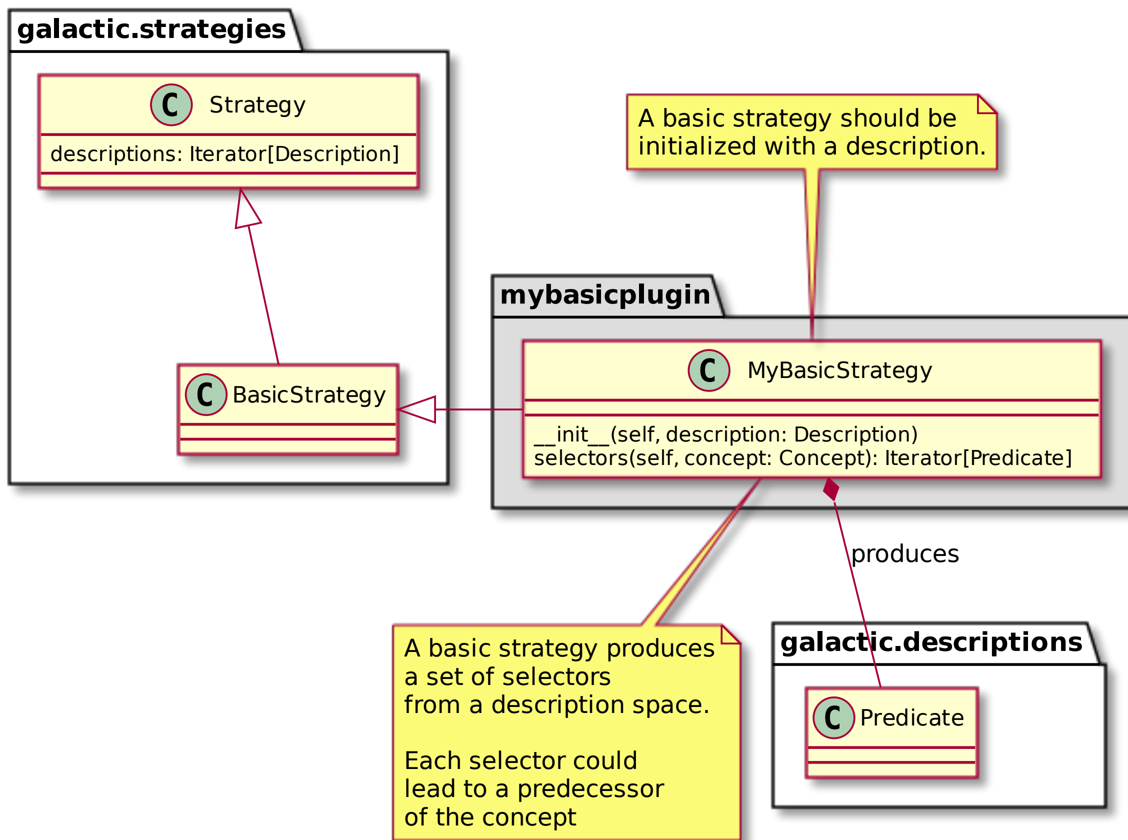


Figure 4: Basic strategy plugin

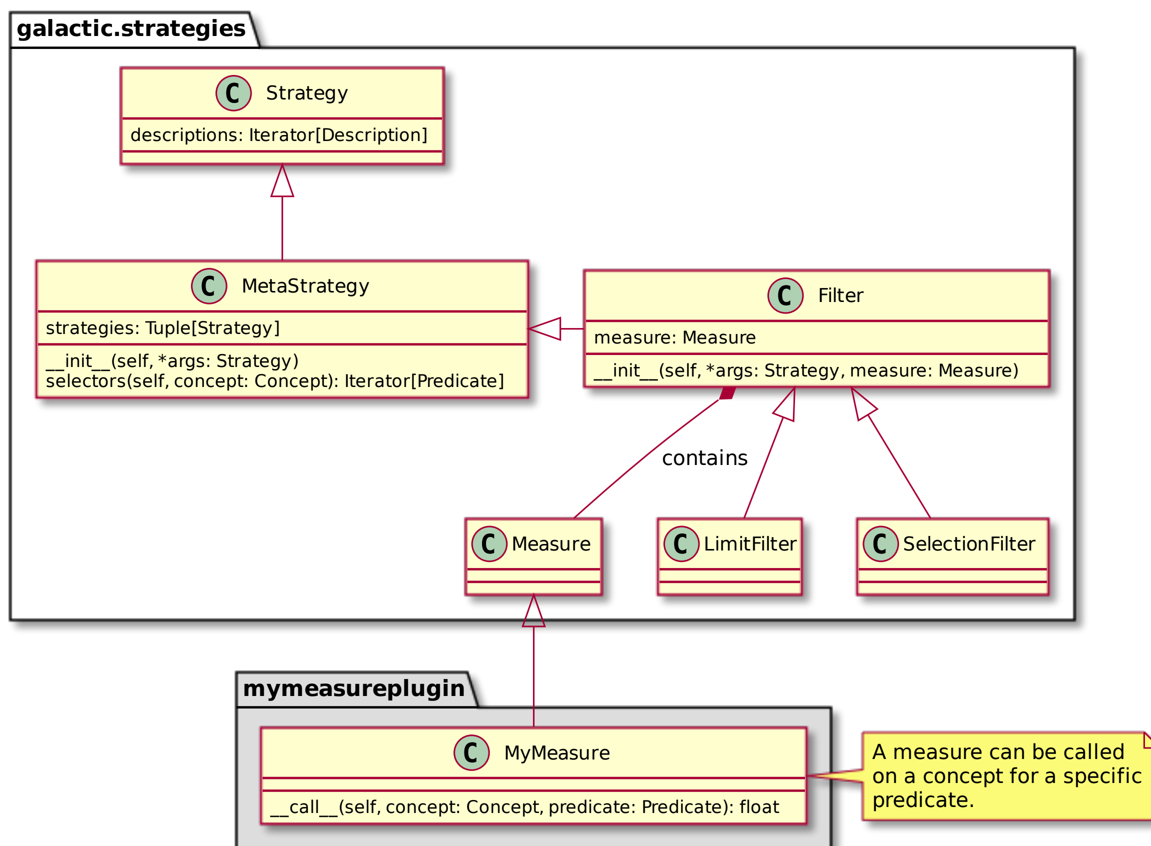


Figure 5: Meta strategy and measure plugins

A meta strategy plugin must declare a class inheriting from the

`galactic.strategies.MetaStrategy`

class and implementing the `selectors` method. This method must produce an iterator over predicates defined by the inner strategies. There are two predefined meta-strategies in the core library:

- `galactic.strategies.LimitFilter`
- `galactic.strategies.SelectionFilter`

which use a measure to choose which predicates to keep. This is the best way to use meta-strategies.

6 Measures

Measures are plugins (cf FIG. 5) of the *py-galactic-core* engine that allow to use the predefined meta-strategies

- `galactic.strategies.LimitFilter`
- `galactic.strategies.SelectionFilter`

A measure plugin must declare a class inheriting from the

galactic.strategies.Measure

class and implementing the `__call__` method. This method must return a float value evaluating a predicate applied to a set of individuals.

```
from typing import Iterable

from galactic.characteristics import Predicate
from galactic.concept import Concept
from galactic.strategies import Measure


class MyMeasure(Measure):
    def __call__(
        self,
        concept: Concept,
        predicate: Predicate = None
    ):
        # must return a float value evaluating the predicate
        # applied on the individuals

def get_classes():
    return {"measure.myplugin.MyMeasure": MyMeasure}
```

References

Demko, Christophe, Karell Bertet, Cyril Faucher, Jean-François Viaud, and Sergei O. Kuznetsov. 2020. 'NextPriorityConcept: A New and Generic Algorithm Computing Concepts from Complex and Heterogeneous Data'. *Theoretical Computer Science* 845: 1–20. https://doi.org/https://doi.org/10.1016/j.tcs.2020.08.026.