argopy Documentation

Release 999

argopy Developers

Apr 22, 2024

GETTING STARTED

argopy is a python library dedicated to *Argo* data access, manipulation and visualisation for standard users as well as Argo experts.

CHAPTER

ONE

DOCUMENTATION

Getting Started

1.1 Installation

1.1.1 Instructions

Install the last release with conda:

```
conda install -c conda-forge argopy
```

or pip:

```
pip install argopy
```

you can also work with the latest dev. version:

```
pip install git+http://github.com/euroargodev/argopy.git@master
```

1.1.2 Required dependencies

- aiohttp
- erddapy
- fsspec < 2023.12.0 (more at #317)
- netCDF4
- scipy
- toolz
- xarray
- requests

Note that Erddapy is required because erddap is the default data fetching backend.

Requirement dependencies details can be found here.

The **argopy** software is continuously tested under latest OS (Linux, Mac OS and Windows) and with python versions 3.8 and 3.9

1.1.3 Optional dependencies

For a complete **argopy** experience, you may also consider to install the following packages:

Utilities

- gsw
- tqdm
- zarr

Performances

- dask
- distributed
- pyarrow

Visualisation

- IPython
- cartopy
- ipykernel
- ipywidgets
- matplotlib
- seaborn

1.2 Usage

To get access to Argo data, all you need is 2 lines of codes:

```
In [1]: from argopy import DataFetcher as ArgoDataFetcher

In [2]: ds = ArgoDataFetcher().region([-75, -45, 20, 30, 0, 100, '2011-01', '2011-06']).
↪to_xarray()
```

In this example, we used a *DataFetcher* to get data for a given space/time region. We retrieved all Argo data measurements from 75W to 45W, 20N to 30N, 0db to 100db and from January to May 2011 (the max date is exclusive). Data are returned as a collection of measurements in a `xarray.Dataset`:

```
In [3]: ds
Out[3]:
<xarray.Dataset>
Dimensions:          (N_POINTS: 9422)
Coordinates:
  * N_POINTS         (N_POINTS) int64 0 1 2 3 4 5 ... 9417 9418 9419 9420 9421
    LATITUDE         (N_POINTS) float64 24.54 24.54 24.54 ... 24.96 24.96 24.96
    LONGITUDE        (N_POINTS) float64 -45.14 -45.14 -45.14 ... -50.4 -50.4
    TIME             (N_POINTS) datetime64[ns] 2011-01-01T11:49:19 ... 2011-0...
Data variables: (12/15)
    CYCLE_NUMBER     (N_POINTS) int64 23 23 23 23 23 23 23 ... 38 38 38 38 38 38
    DATA_MODE        (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D' 'D'
    DIRECTION        (N_POINTS) <U1 'A' 'A' 'A' 'A' 'A' ... 'A' 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER  (N_POINTS) int64 1901463 1901463 ... 1901463 1901463
    POSITION_QC      (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    PRES             (N_POINTS) float64 5.0 10.0 15.0 20.0 ... 90.0 95.0 100.0
    ...               ...
    PSAL_ERROR       (N_POINTS) float32 0.01 0.01 0.01 ... 0.01017 0.01016
    PSAL_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TEMP             (N_POINTS) float64 24.08 24.08 24.09 ... 21.54 21.28 21.19
    TEMP_ERROR       (N_POINTS) float32 0.002 0.002 0.002 ... 0.002 0.002 0.002
    TEMP_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TIME_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
Attributes:
    DATA_ID:              ARGO
    DOI:                  http://doi.org/10.17882/42182
    Fetched_from:         https://erddap.ifremer.fr/erddap
    Fetched_by:           docs
    Fetched_date:         2024/04/22
    Fetched_constraints:  [x=-75.00/-45.00; y=20.00/30.00; z=0.0/100.0; t=201...
    Fetched_uri:          ['https://erddap.ifremer.fr/erddap/tabledap/ArgoFlo...
    history:              Variables filtered according to DATA_MODE; Variable...
```

Fetched data are returned as a 1D array collection of measurements. If you prefer to work with a 2D array collection of vertical profiles, simply transform the dataset with the `xarray.Dataset` accessor method *Dataset.argo.point2profile()*:

```
In [4]: ds = ds.argo.point2profile()

In [5]: ds
```

(continues on next page)

(continued from previous page)

```
Out[5]:
<xarray.Dataset>
Dimensions:          (N_PROF: 469, N_LEVELS: 55)
Coordinates:
  * N_PROF           (N_PROF) int64 15 286 125 0 223 46 ... 371 461 467 433 30
  * N_LEVELS         (N_LEVELS) int64 0 1 2 3 4 5 6 7 ... 48 49 50 51 52 53 54
    LATITUDE         (N_PROF) float64 24.54 25.04 21.48 ... 21.49 26.67 24.96
    LONGITUDE        (N_PROF) float64 -45.14 -51.58 -60.82 ... -66.83 -50.4
    TIME             (N_PROF) datetime64[ns] 2011-01-01T11:49:19 ... 2011-05-...
Data variables: (12/15)
    CYCLE_NUMBER     (N_PROF) int64 23 10 135 23 119 160 4 ... 163 1 5 2 10 38
    DATA_MODE        (N_PROF) <U1 'D' 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D' 'D'
    DIRECTION        (N_PROF) <U1 'A' 'A' 'A' 'A' 'A' ... 'A' 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER  (N_PROF) int64 1901463 4901211 4900818 ... 6900778 1901463
    POSITION_QC      (N_PROF) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1 1
    PRES             (N_PROF, N_LEVELS) float64 5.0 10.0 15.0 ... nan nan nan
    ...               ...
    PSAL_ERROR       (N_PROF, N_LEVELS) float32 0.01 0.01 0.01 ... nan nan nan
    PSAL_QC          (N_PROF) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1 1
    TEMP             (N_PROF, N_LEVELS) float64 24.08 24.08 24.09 ... nan nan
    TEMP_ERROR       (N_PROF) float32 0.002 0.002 0.002 ... 0.002 0.0025 0.002
    TEMP_QC          (N_PROF) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1 1
    TIME_QC          (N_PROF) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1 1
Attributes:
    DATA_ID:              ARGO
    DOI:                  http://doi.org/10.17882/42182
    Fetched_from:         https://erddap.ifremer.fr/erddap
    Fetched_by:           docs
    Fetched_date:         2024/04/22
    Fetched_constraints:  [x=-75.00/-45.00; y=20.00/30.00; z=0.0/100.0; t=201...
    Fetched_uri:          ['https://erddap.ifremer.fr/erddap/tabledap/ArgoFlo...
    history:              Variables filtered according to DATA_MODE; Variable...
```

You can also fetch data for a specific float using its WMO number:

```
In [6]: ds = ArgoDataFetcher().float(6902746).to_xarray()
```

or for a float profile using the cycle number:

```
In [7]: ds = ArgoDataFetcher().profile(6902755, 12).to_xarray()
```

1.3 Why argopy ?

Surprisingly, the Argo community never provided its user base with a Python software to easily access and manipulate Argo measurements: **argopy** aims to fill this gap.

Despite, or because, its tremendous success in data management and in developping good practices and well calibrated procedures [ADMT], the Argo dataset is very complex: with thousands of different variables, tens of reference tables and a user manual more than 100 pages long: **argopy** aims to help you navigate this complex realm.

For non-experts of the Argo dataset, it has become rather complicated to get access to Argo measurements. This is mainly due to:

- Argo measurements coming from many different models of floats or sensors,
- quality control of *in situ* measurements of autonomous platforms being really a matter of ocean and data experts,
- the Argo data management workflow being distributed between more than 10 Data Assembly Centers all around the world.

1.3.1 Less data wrangling, more scientific analysis

In order to ease Argo data analysis for the vast majority of **standard** users, we implemented in **argopy** different levels of verbosity and data processing to hide or simply remove variables only meaningful to **experts**. Let **argopy** manage data wrangling, and focus on your scientific analysis.

If you don't know in which category you would place yourself, try to answer the following questions:

- [ ] what is a WMO number ?
- [ ] what is the difference between Delayed and Real Time data mode ?
- [ ] what is an adjusted parameter ?
- [ ] what a QC flag of 3 means ?

If you don't answer to more than 1 question: you probably will feel more comfortable with the *standard* or *research* user modes.

By default, all **argopy** data fetchers are set to work with a **standard** user mode, the other possible modes are **research** and **expert**.

Each user modes and how to select it are further explained in the dedicated documentation section: *User mode (, ,)*.

1.4 What is Argo ?

Argo is a real-time global ocean in situ observing system.

The ocean is a key component of the Earth climate system. It thus needs a continuous real-time monitoring to help scientists better understand its dynamic and predict its evolution. All around the world, oceanographers have managed to join their efforts and set up a Global Ocean Observing System among which *Argo* is a key component.

Argo is a global network of nearly 4000 autonomous probes measuring pressure, temperature and salinity from the surface to 2000m depth every 10 days. The localisation of these probes is nearly random between the 60th parallels (see live coverage here). All probes data are collected by satellite in real-time, processed by several data centers and finally merged in a single dataset (collecting more than 2 millions of vertical profiles data) made freely available to anyone through a ftp server or monthly zip snapshots.

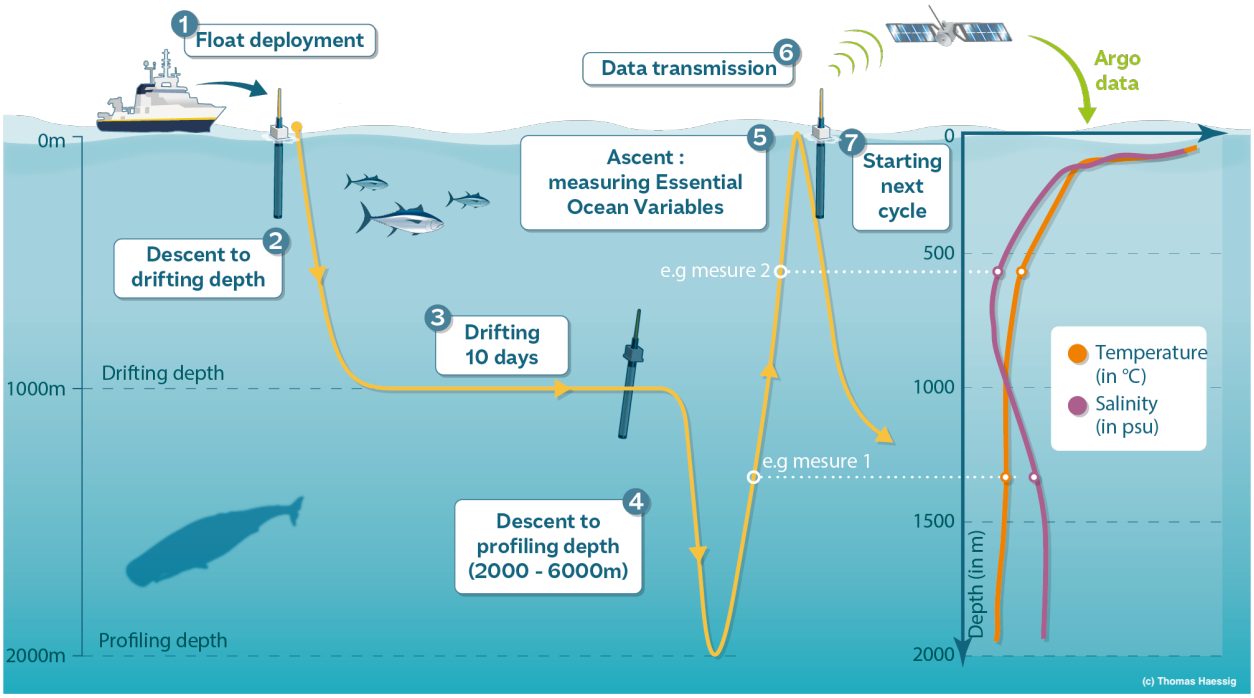
The Argo international observation array was initiated in 1999 and soon revolutionized our perspective on the large scale structure and variability of the ocean by providing seasonally and regionally unbiased in situ temperature/salinity measurements of the ocean interior, key information that satellites can't provide (Riser et al, 2016).

The Argo array reached its full global coverage (of 1 profile per month and per 3x3 degree horizontal area) in 2007, and continuously pursues its evolution to fulfill new scientific requirements (Roemmich et al, 2019). It now extents to higher latitudes and some of the floats are able to profile down to 4000m and 6000m. New floats are also equipped with biogeochemical sensors, measuring oxygen and chlorophyll for instance. Argo is thus providing a deluge of in situ data: more than 400 profiles per day.

Each Argo probe is an autonomous, free drifting, profiling float, i.e. a probe that can't control its trajectory but is able to control its buoyancy and thus to move up and down the water column as it wishes. Argo floats continuously operate the same program, or cycle, illustrated in the figure below. After 9 to 10 days of free drift at a parking depth of about 1000m, a typical Argo float dives down to 2000m and then shoals back to the surface while measuring pressure,

temperature and salinity. Once it reaches the surface, the float sends by satellite its measurements to a data center where they are processed in real time and made freely available on the web in less than 24h00.

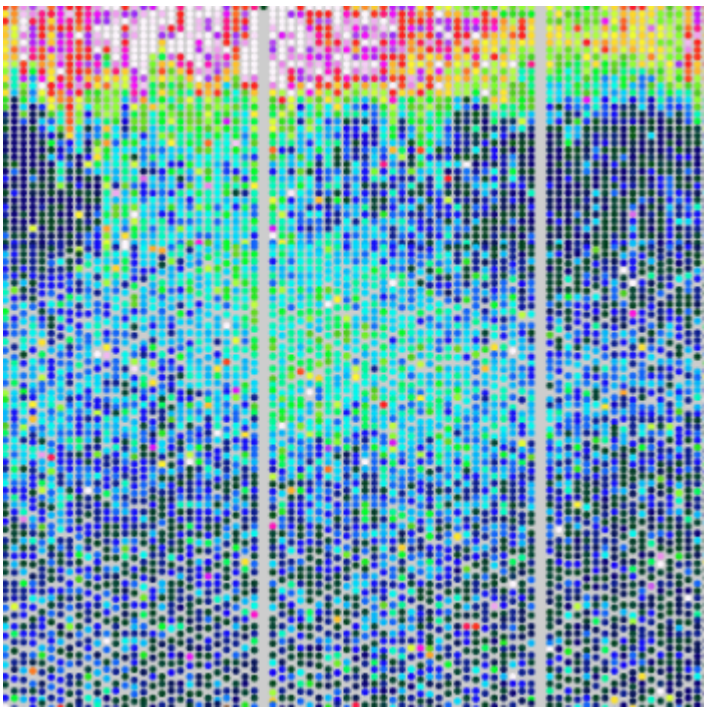
Typical 10 days program, cycle, of an Argo float:



1.5 Gallery

Here's a list of examples on how to use **argopy**. We will be adding more examples soon. Contributions are highly welcomed and appreciated. So, if you are interested in contributing, please consult the *Contributing to argopy* guide.

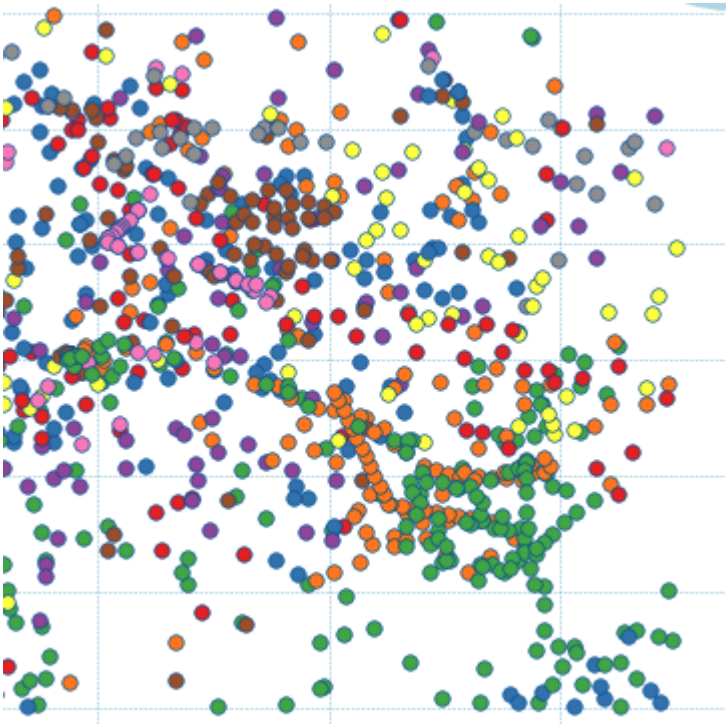
1.5.1 Notebook Examples



BGC one float data A notebook to download and plot one BGC float data

Online viewer

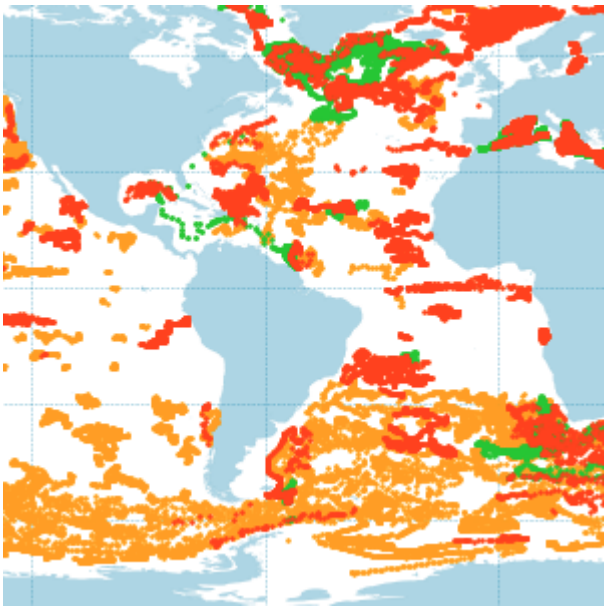
Download notebook



BGC regional data A notebook to download and plot BGC data in a specific ocean region

Online viewer

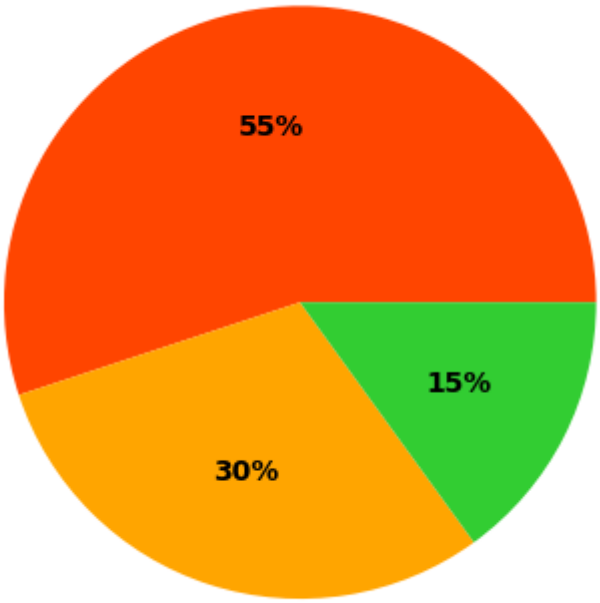
Download notebook



Scatter map with data mode of one BGC variable A notebook to plot a global map where profile locations are color coded with one BGC parameter data mode

Online viewer

Download notebook



BGC data mode census A notebook to make a global census of all BGC parameter data mode and a pie plot with results

Online viewer

Download notebook

Notebook tags Legend

Data selection
: region, : float, : profile

Dataset
+ : core+deep, : BGC

User mode
: expert, : standard, : research

Data sources
: erddap, : gdac, : argovis

User Guide

- *Fetching Argo data*
- *Working with Argo data*
- *Argo meta-data*
- *Performances*

1.6 Fetching Argo data

To fetch (i.e. access, download, format) Argo data, **argopy** provides the `DataFetcher` class. In this section of the documentation, we explain how to use it.

Several `DataFetcher` arguments exist to help you select the *dataset*, the *data source* and the *user mode* the most suited for your applications; and also to improve *performances*.

You define the selection of data you want to fetch with one of the `DataFetcher` methods: *region*, *float* or *profile*.

These methods and arguments are all explained in the following sections:

- *Data selection*
- *Data sources*
- *Dataset*
- *User mode (, ,)*

1.6.1 Data selection

To access Argo data with a `DataFetcher`, you need to define how to select your data of interest.

argopy provides 3 different data selection methods:

- *For a space/time domain*,
- *For one or more floats*,
- *For one or more profiles*.

To show how these methods (i.e. *access points*) work, let's first create a `DataFetcher`:

```
In [1]: import argopy

In [2]: f = argopy.DataFetcher()

In [3]: f
Out[3]:
<datafetcher.erddap> 'No access point initialised'
Available access points: float, profile, region
Performances: cache=False, parallel=False
User mode: standard
Dataset: phy
```

By default, **argopy** will load the `phy` dataset (*see here for details*), in `standard` user mode (*see here for details*) from the `erddap` data source (*see here for details*).

The standard `DataFetcher` print indicates all available access points, and here, that none is selected yet.

For a space/time domain

Use the fetcher access point *argopy.DataFetcher.region()* to select data for a *rectangular* space/time domain. For instance, to retrieve data from 75W to 45W, 20N to 30N, 0db to 10db and from January to May 2011:

```
In [4]: f = f.region([-75, -45, 20, 30, 0, 10, '2011-01', '2011-06'])

In [5]: f.data
Out[5]:
<xarray.Dataset>
Dimensions:          (N_POINTS: 998)
Coordinates:
  * N_POINTS         (N_POINTS) int64 0 1 2 3 4 5 6 ... 992 993 994 995 996 997
    LATITUDE         (N_POINTS) float64 24.54 24.54 25.04 ... 26.67 24.96 24.96
    LONGITUDE        (N_POINTS) float64 -45.14 -45.14 -51.58 ... -50.4 -50.4
    TIME             (N_POINTS) datetime64[ns] 2011-01-01T11:49:19 ... 2011-0...
Data variables: (12/15)
    CYCLE_NUMBER     (N_POINTS) int64 23 23 10 10 10 10 10 ... 1 5 2 10 10 38 38
    DATA_MODE        (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D' 'D'
    DIRECTION        (N_POINTS) <U1 'A' 'A' 'A' 'A' 'A' ... 'A' 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER  (N_POINTS) int64 1901463 1901463 ... 1901463 1901463
    POSITION_QC      (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    PRES             (N_POINTS) float64 5.0 10.0 2.0 4.0 ... 5.12 9.42 5.0 10.0
    ...                  ...
    PSAL_ERROR       (N_POINTS) float32 0.01 0.01 0.01 ... 0.01 0.01091 0.01182
    PSAL_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TEMP             (N_POINTS) float64 24.08 24.08 24.03 ... 25.64 25.1 24.79
    TEMP_ERROR       (N_POINTS) float32 0.002 0.002 0.002 ... 0.0025 0.002 0.002
    TEMP_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TIME_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
Attributes:
    DATA_ID:              ARGO
    DOI:                  http://doi.org/10.17882/42182
    Fetched_from:         https://erddap.ifremer.fr/erddap
    Fetched_by:           docs
    Fetched_date:         2024/04/22
    Fetched_constraints:  [x=-75.00/-45.00; y=20.00/30.00; z=0.0/10.0; t=2011...
    Fetched_uri:          ['https://erddap.ifremer.fr/erddap/tabledap/ArgoFlo...
    history:              Variables filtered according to DATA_MODE; Variable...
```

You can now see that the standard *DataFetcher* print has been updated with information for the data selection.

Note:

- The constraint on time is not mandatory: if not specified, the fetcher will return all data available in this region.
- The last time bound is exclusive: that's why here we specify June to retrieve data collected in May.

For one or more floats

If you know the Argo float unique identifier number called a WMO number you can use the fetcher access point *DataFetcher.float()* to specify one or more float WMO platform numbers to select.

For instance, to select data for float WMO *6902746*:

```
In [6]: f = f.float(6902746)

In [7]: f.data
Out[7]:
<xarray.Dataset>
Dimensions:          (N_POINTS: 12518)
Coordinates:
  * N_POINTS         (N_POINTS) int64 0 1 2 3 4 ... 12514 12515 12516 12517
    LATITUDE         (N_POINTS) float64 20.08 20.08 20.08 ... 16.67 16.67 16.67
    LONGITUDE        (N_POINTS) float64 -60.17 -60.17 -60.17 ... -77.13 -77.13
    TIME             (N_POINTS) datetime64[ns] 2017-07-06T14:49:00 ... 2020-0...
Data variables: (12/15)
    CYCLE_NUMBER     (N_POINTS) int64 1 1 1 1 1 1 1 ... 117 117 117 117 117 117
    DATA_MODE        (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D' 'D'
    DIRECTION        (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'A' 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER  (N_POINTS) int64 6902746 6902746 ... 6902746 6902746
    POSITION_QC      (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    PRES             (N_POINTS) float64 9.0 14.0 24.0 ... 1.514e+03 1.526e+03
    ...               ...
    PSAL_ERROR       (N_POINTS) float64 0.01003 0.01003 0.01003 ... 0.01 0.01
    PSAL_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TEMP             (N_POINTS) float64 28.04 28.03 28.02 ... 4.299 4.254 4.238
    TEMP_ERROR       (N_POINTS) float64 0.002 0.002 0.002 ... 0.002 0.002 0.002
    TEMP_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TIME_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
Attributes:
    DATA_ID:              ARGO
    DOI:                  http://doi.org/10.17882/42182
    Fetched_from:         https://erddap.ifremer.fr/erddap
    Fetched_by:           docs
    Fetched_date:         2024/04/22
    Fetched_constraints:  WMO6902746
    Fetched_uri:          ['https://erddap.ifremer.fr/erddap/tabledap/ArgoFlo...
    history:              Variables filtered according to DATA_MODE; Variable...
```

To fetch data for a collection of floats, input them in a list:

```
In [8]: f = f.float([6902746, 6902755])

In [9]: f.data
Out[9]:
<xarray.Dataset>
Dimensions:          (N_POINTS: 31289)
Coordinates:
  * N_POINTS         (N_POINTS) int64 0 1 2 3 4 ... 31285 31286 31287 31288
    LATITUDE         (N_POINTS) float64 20.08 20.08 20.08 ... 43.81 43.81 43.81
    LONGITUDE        (N_POINTS) float64 -60.17 -60.17 -60.17 ... -28.85 -28.85
```

(continues on next page)

(continued from previous page)

```
    TIME             (N_POINTS) datetime64[ns] 2017-07-06T14:49:00 ... 2023-0...
Data variables: (12/15)
    CYCLE_NUMBER     (N_POINTS) int64 1 1 1 1 1 1 1 ... 177 177 177 177 177 177
    DATA_MODE        (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'A' 'A' 'A' 'A' 'A'
    DIRECTION        (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'A' 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER  (N_POINTS) int64 6902746 6902746 ... 6902755 6902755
    POSITION_QC      (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    PRES             (N_POINTS) float64 9.0 14.0 24.0 34.0 ... 278.0 285.0 296.0
    ...               ...
    PSAL_ERROR       (N_POINTS) float32 0.01003 0.01003 0.01003 ... nan nan nan
    PSAL_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TEMP             (N_POINTS) float64 28.04 28.03 28.02 ... 13.5 13.45 13.49
    TEMP_ERROR       (N_POINTS) float32 0.002 0.002 0.002 0.002 ... nan nan nan
    TEMP_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TIME_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
Attributes:
    DATA_ID:              ARGO
    DOI:                  http://doi.org/10.17882/42182
    Fetched_from:         https://erddap.ifremer.fr/erddap
    Fetched_by:           docs
    Fetched_date:         2024/04/22
    Fetched_constraints:  WMO6902746;WMO6902755
    Fetched_uri:          ['https://erddap.ifremer.fr/erddap/tabledap/ArgoFlo...
    history:              Variables filtered according to DATA_MODE; Variable...
```

For one or more profiles

Use the fetcher access point *argopy.DataFetcher.profile()* to specify the float WMO platform number and the profile cycle number(s) to retrieve profiles for.

For instance, to retrieve data for the 12th profile of float WMO 6902755:

```
In [10]: f = f.profile(6902755, 12)

In [11]: f.data
Out[11]:
<xarray.Dataset>
Dimensions:           (N_POINTS: 107)
Coordinates:
  * N_POINTS          (N_POINTS) int64 0 1 2 3 4 5 6 ... 101 102 103 104 105 106
    LATITUDE          (N_POINTS) float64 63.68 63.68 63.68 ... 63.68 63.68 63.68
    LONGITUDE         (N_POINTS) float64 -28.81 -28.81 -28.81 ... -28.81 -28.81
    TIME              (N_POINTS) datetime64[ns] 2018-10-19T23:52:00 ... 2018-1...
Data variables: (12/15)
    CYCLE_NUMBER      (N_POINTS) int64 12 12 12 12 12 12 12 ... 12 12 12 12 12 12
    DATA_MODE         (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D' 'D'
    DIRECTION         (N_POINTS) <U1 'A' 'A' 'A' 'A' 'A' ... 'A' 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER   (N_POINTS) int64 6902755 6902755 ... 6902755 6902755
    POSITION_QC       (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    PRES              (N_POINTS) float64 3.0 4.0 5.0 ... 1.713e+03 1.732e+03
    ...                ...
```

(continues on next page)

(continued from previous page)

```
    PSAL_ERROR       (N_POINTS) float64 0.01 0.01 0.01 0.01 ... 0.01 0.01 0.01
    PSAL_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TEMP             (N_POINTS) float64 7.598 7.599 7.602 ... 3.588 3.549 3.536
    TEMP_ERROR       (N_POINTS) float64 0.002 0.002 0.002 ... 0.002 0.002 0.002
    TEMP_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TIME_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
Attributes:
    DATA_ID:              ARGO
    DOI:                  http://doi.org/10.17882/42182
    Fetched_from:         https://erddap.ifremer.fr/erddap
    Fetched_by:           docs
    Fetched_date:         2024/04/22
    Fetched_constraints:  WMO6902755_CYC12
    Fetched_uri:          ['https://erddap.ifremer.fr/erddap/tabledap/ArgoFlo...
    history:              Variables filtered according to DATA_MODE; Variable...
```

To fetch data for more than one profile, input them in a list:

```
In [12]: f = f.profile(6902755, [3, 12])

In [13]: f.data
Out[13]:
<xarray.Dataset>
Dimensions:          (N_POINTS: 215)
Coordinates:
  * N_POINTS         (N_POINTS) int64 0 1 2 3 4 5 6 ... 209 210 211 212 213 214
    LATITUDE         (N_POINTS) float64 59.72 59.72 59.72 ... 63.68 63.68 63.68
    LONGITUDE        (N_POINTS) float64 -31.24 -31.24 -31.24 ... -28.81 -28.81
    TIME             (N_POINTS) datetime64[ns] 2018-07-22T00:03:00 ... 2018-1...
Data variables: (12/15)
    CYCLE_NUMBER     (N_POINTS) int64 3 3 3 3 3 3 3 3 ... 12 12 12 12 12 12 12
    DATA_MODE        (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D' 'D'
    DIRECTION        (N_POINTS) <U1 'A' 'A' 'A' 'A' 'A' ... 'A' 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER  (N_POINTS) int64 6902755 6902755 ... 6902755 6902755
    POSITION_QC      (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    PRES             (N_POINTS) float64 3.0 4.0 5.0 ... 1.713e+03 1.732e+03
    ...               ...
    PSAL_ERROR       (N_POINTS) float64 0.01 0.01 0.01 0.01 ... 0.01 0.01 0.01
    PSAL_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TEMP             (N_POINTS) float64 8.742 8.743 8.744 ... 3.588 3.549 3.536
    TEMP_ERROR       (N_POINTS) float64 0.002 0.002 0.002 ... 0.002 0.002 0.002
    TEMP_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TIME_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
Attributes:
    DATA_ID:              ARGO
    DOI:                  http://doi.org/10.17882/42182
    Fetched_from:         https://erddap.ifremer.fr/erddap
    Fetched_by:           docs
    Fetched_date:         2024/04/22
    Fetched_constraints:  WMO6902755_CYC3_CYC12
    Fetched_uri:          ['https://erddap.ifremer.fr/erddap/tabledap/ArgoFlo...
    history:              Variables filtered according to DATA_MODE; Variable...
```

Note: You can chain data selection and fetching in a single command line:

```
f = argopy.DataFetcher().region([-75, -45, 20, 30, 0, 10, '2011-01-01', '2011-06']).
↪load()
f.data
```

1.6.2 Data sources

Hint: **argopy** can fetch data from several data sources. To make sure you understand where you're getting data from, have a look at this section.

Contents

Let's start with standard import:

```
In [1]: import argopy

In [2]: from argopy import DataFetcher as ArgoDataFetcher

In [3]: argopy.reset_options()
```

Available data sources

argopy can get access to Argo data from the following sources:

1. **the Ifremer erddap server (Default).**
   The erddap server database is updated daily and doesn't require you to download anymore data than what you need. You can select this data source with the keyword `erddap` and methods described below. The Ifremer erddap dataset is based on mono-profile files of the GDAC. Since this is the most efficient method to fetcher Argo data, it's the default data source in **argopy**.

2. **an Argo GDAC server or any other GDAC-compliant local folder.**
   You can fetch data from any of the 3 official GDAC online servers: the Ifremer https and ftp and the US ftp. This data source can also point toward your own local copy of the GDAC ftp content. You can select this data source with the keyword `gdac` and methods described below.

3. **the Argovis server.**
   The Argovis server database is updated daily and only provides access to curated Argo data (QC=1 only). You can select this data source with the keyword `argovis` and methods described below.

Selecting a source

You have several ways to specify which data source you want to use:

- **using argopy global options**:

```
In [4]: argopy.set_options(src='erddap')
Out[4]: <argopy.options.set_options at 0x7fd7126fbe80>
```

- **in a temporary context**:

```
In [5]: with argopy.set_options(src='erddap'):
   ...:     loader = ArgoDataFetcher().profile(6902746, 34)
   ...:
```

- **with an argument in the data fetcher**:

```
In [6]: loader = ArgoDataFetcher(src='erddap').profile(6902746, 34)
```

Comparing data sources

Features

Each of the data sources have their own features and capabilities. Here is a summary:

Table 1: Table of **argopy** data sources features

		erddap	gdac	argovis
Access Points:				
	region	X	X	X
	float	X	X	X
	profile	X	X	X
User mode:				
	expert	X	X	
	standard	X	X	X
	research	X	X	
Dataset:				
	core (T/S)	X	X	X
	BGC	X	X	
	Deep	X	X	X
	Trajectories			
	Reference data for DMQC	X		

Fetched data and variables

You may wonder if the fetched data are different from the available data sources.
This will depend on the last update of each data sources and of your local data.

GDAC ftp

erddap

argovis

Let's retrieve one float data from a local sample of the GDAC ftp (a sample GDAC ftp is downloaded automatically with the method *argopy.tutorial.open_dataset()*):

```
# Download ftp sample and get the ftp local path:
In [7]: ftproot = argopy.tutorial.open_dataset('gdac')[0]

# then fetch data:
In [8]: with argopy.set_options(src='gdac', ftp=ftproot):
   ...:     ds = ArgoDataFetcher().float(1900857).load().data
   ...:     print(ds)
   ...:
<xarray.Dataset>
Dimensions:          (N_POINTS: 20966)
Coordinates:
  * N_POINTS         (N_POINTS) int64 0 1 2 3 4 ... 20962 20963 20964 20965
    TIME             (N_POINTS) datetime64[ns] 2008-02-25T04:03:00 ... 2013-0...
    LATITUDE         (N_POINTS) float64 -39.93 -39.93 -39.93 ... -44.16 -44.16
    LONGITUDE        (N_POINTS) float64 10.81 10.81 10.81 ... 92.65 92.65 92.65
Data variables: (12/15)
    CYCLE_NUMBER     (N_POINTS) int64 0 0 0 0 0 0 0 ... 192 192 192 192 192 192
    DATA_MODE        (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D' 'D'
    DIRECTION        (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'A' 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER  (N_POINTS) int64 1900857 1900857 ... 1900857 1900857
    POSITION_QC      (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    PRES             (N_POINTS) float64 17.0 25.0 35.0 ... 1.964e+03 1.987e+03
    ...               ...
    PSAL_ERROR       (N_POINTS) float64 0.02 0.02 0.02 0.02 ... 0.02 0.02 0.02
    PSAL_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TEMP             (N_POINTS) float64 16.14 16.14 16.03 ... 2.431 2.422 2.413
    TEMP_ERROR       (N_POINTS) float64 0.002 0.002 0.002 ... 0.002 0.002 0.002
    TEMP_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TIME_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
Attributes:
    DATA_ID:              ARGO
    DOI:                  http://doi.org/10.17882/42182
    Fetched_from:         /home/docs/.argopy_tutorial_data/ftp
    Fetched_by:           docs
    Fetched_date:         2024/04/22
    Fetched_constraints:  WMO1900857
    Fetched_uri:          /home/docs/.argopy_tutorial_data/ftp/dac/coriolis/1...
    history:              Variables filtered according to DATA_MODE; Variable...
```

Let's now retrieve the latest data for this float from the `erddap`:

```
In [9]: with argopy.set_options(src='erddap'):
   ...:     ds = ArgoDataFetcher().float(1900857).load().data
   ...:     print(ds)
   ...:
<xarray.Dataset>
Dimensions:          (N_POINTS: 20966)
Coordinates:
  * N_POINTS         (N_POINTS) int64 0 1 2 3 4 ... 20962 20963 20964 20965
    LATITUDE         (N_POINTS) float64 -39.93 -39.93 -39.93 ... -44.16 -44.16
    LONGITUDE        (N_POINTS) float64 10.81 10.81 10.81 ... 92.65 92.65 92.65
    TIME             (N_POINTS) datetime64[ns] 2008-02-25T04:03:00 ... 2013-0...
Data variables: (12/15)
    CYCLE_NUMBER     (N_POINTS) int64 0 0 0 0 0 0 0 ... 192 192 192 192 192 192
    DATA_MODE        (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D' 'D'
    DIRECTION        (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'A' 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER  (N_POINTS) int64 1900857 1900857 ... 1900857 1900857
    POSITION_QC      (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    PRES             (N_POINTS) float64 17.0 25.0 35.0 ... 1.964e+03 1.987e+03
    ...               ...
    PSAL_ERROR       (N_POINTS) float64 0.02 0.02 0.02 0.02 ... 0.02 0.02 0.02
    PSAL_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TEMP             (N_POINTS) float64 16.14 16.14 16.03 ... 2.431 2.422 2.413
    TEMP_ERROR       (N_POINTS) float64 0.002 0.002 0.002 ... 0.002 0.002 0.002
    TEMP_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TIME_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
Attributes:
    DATA_ID:              ARGO
    DOI:                  http://doi.org/10.17882/42182
    Fetched_from:         https://erddap.ifremer.fr/erddap
    Fetched_by:           docs
    Fetched_date:         2024/04/22
    Fetched_constraints:  WMO1900857
    Fetched_uri:          ['https://erddap.ifremer.fr/erddap/tabledap/ArgoFlo...
    history:              Variables filtered according to DATA_MODE; Variable...
```

And with argovis:

```
In [10]: with argopy.set_options(src='argovis'):
   ....:     ds = ArgoDataFetcher().float(1900857).load().data
   ....:     print(ds)
   ....:
<xarray.Dataset>
Dimensions:          (N_POINTS: 21029)
Coordinates:
  * N_POINTS         (N_POINTS) int64 0 1 2 3 4 ... 21025 21026 21027 21028
    TIME             (N_POINTS) datetime64[ns] 2008-02-28T01:23:00 ... 2013-0...
    LATITUDE         (N_POINTS) float64 -40.02 -40.02 -40.02 ... -44.16 -44.16
    LONGITUDE        (N_POINTS) float64 10.54 10.54 10.54 ... 92.65 92.65 92.65
Data variables:
    CYCLE_NUMBER     (N_POINTS) int64 0 0 0 0 0 0 0 ... 192 192 192 192 192 192
    DATA_MODE        (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D' 'D'
    DIRECTION        (N_POINTS) <U1 'A' 'A' 'A' 'A' 'A' ... 'A' 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER  (N_POINTS) int64 1900857 1900857 ... 1900857 1900857
```

(continues on next page)

(continued from previous page)

```
    POSITION_QC      (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1 1
    PRES             (N_POINTS) int64 16 26 37 45 55 ... 1913 1938 1964 1987
    PSAL             (N_POINTS) float64 34.74 34.73 34.67 ... 34.71 34.71 34.72
    TEMP             (N_POINTS) float64 16.69 16.59 15.92 ... 2.431 2.422 2.413
    TIME_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1 1
Attributes:
    DATA_ID:              ARGO
    DOI:                  http://doi.org/10.17882/42182
    Fetched_from:         https://argovisbeta02.colorado.edu
    Fetched_by:           docs
    Fetched_date:         2024/04/22
    Fetched_constraints:  WMO1900857
    Fetched_uri:          ['https://argovisbeta02.colorado.edu/catalog/platfo...
```

Status of sources

With remote, online data sources, it may happens that the data server is experiencing down time. With local data sources, the availability of the path is checked when it is set. But it may happens that the path points to a disk that get unmounted or unplugged after the option setting.

If you're running your analysis on a Jupyter notebook, you can use the *argopy.status()* method to insert a data status monitor on a cell output. All available data sources will be monitored continuously.

```
argopy.status()
```

If one of the data source become unavailable, you will see the status bar changing to something like:

Note that the *argopy.status()* method has a `refresh` option to let you specify the refresh rate in seconds of the monitoring.

Last, you can check out the following argopy status webpage that monitors all important resources to the software.

Setting-up your own local copy of the GDAC ftp

Data fetching with the `gdac` data source will require you to specify the path toward your local copy of the GDAC ftp server with the `ftp` option.

This is not an issue for expert users, but standard users may wonder how to set this up. The primary distribution point for Argo data, the only one with full support from data centers and with nearly a 100% time availability, is the GDAC ftp. Two mirror servers are available:

- France Coriolis: ftp://ftp.ifremer.fr/ifremer/argo
- US GODAE: ftp://usgodae.org/pub/outgoing/argo

If you want to get your own copy of the ftp server content, you have 2 options detailed below.

Copy with DOI reference

If you need an Argo database referenced with a DOI, one that you could use to make your analysis reproducible, then we recommend you to visit https://doi.org/10.17882/42182. There, you will find links toward monthly snapshots of the Argo database, and each snapshot has its own DOI.

For instance, https://doi.org/10.17882/42182#92121 points toward the snapshot archived on February 10st 2022. Simply download the tar archive file (about 44Gb) and uncompress it locally.

You're done !

Synchronized copy

If you need a local Argo database always up to date with the GDAC server, Ifremer provides a nice rsync service. The rsync server "vdmzrs.ifremer.fr" provides a synchronization service between the "dac" directory of the GDAC and a user mirror. The "dac" index files are also available from "argo-index".

From the user side, the rsync service:

- Downloads the new files
- Downloads the updated files
- Removes the files that have been removed from the GDAC
- Compresses/uncompresses the files during the transfer
- Preserves the files creation/update dates
- Lists all the files that have been transferred (easy to use for a user side post-processing)

To synchronize the whole dac directory of the Argo GDAC:

```
rsync -avzh --delete vdmzrs.ifremer.fr::argo/ /home/mydirectory/...
```

To synchronize the index:

```
rsync -avzh --delete vdmzrs.ifremer.fr::argo-index/ /home/mydirectory/...
```

Note: The first synchronisation of the whole dac directory of the Argo GDAC (365Gb) can take quite a long time (several hours).

1.6.3 Dataset

Hint: **argopy** makes a difference between the physical and biogeochemical parameters. To make sure you understand which data you're getting, have a look at this section.

Contents

Argo data are distributed as a single dataset. It is referenced at https://doi.org/10.17882/42182.

But they are several Argo missions with specific files and parameters that need special handling by **argopy**, namely:

- the core Argo Mission: from floats that measure temperature, salinity, pressure down to 2000m,
- the Deep Argo Mission: from floats that measure temperature, salinity, pressure down to 6000m,
- and the BGC-Argo Mission: from floats that measure temperature, salinity, pressure and oxygen, pH, nitrate, chlorophyll, backscatter, irradiance down to 2000m.

Argo dataset available in argopy

In **argopy** we simply make the difference between physical and biogeochemical parameters in the Argo dataset. This is because the Deep Argo mission data are accessible following the same files and parameters than those from the Core mission. Only BGC-Argo data requires specific files and parameters.

In **argopy** you can thus get access to the following Argo data:

1. **the phy dataset, for *physical* parameters.**
   This dataset provides data from floats that measure temperature, salinity, pressure, without limitation in depth. It is available from all *Available data sources*. Since this is the most common Argo data subset it's selected with the `phy` keyword by default in **argopy**.
2. **the bgc dataset, for *biogeochemical* parameters.**
   This dataset provides data from floats that measure temperature, salinity, pressure and oxygen, pH, nitrate, chlorophyll, backscatter, irradiance, without limitation in depth. You can select this dataset with the keyword `bgc` and methods described below.

Selecting a dataset

You have several ways to specify which dataset you want to use:

- **using argopy global options**:

```
In [1]: import argopy

In [2]: argopy.set_options(dataset='bgc')
Out[2]: <argopy.options.set_options at 0x7fd6d9a6b5e0>
```

- **with an option in a temporary context**:

```
In [3]: import argopy

In [4]: with argopy.set_options(dataset='phy'):
```

(continues on next page)

(continued from previous page)

```
...:     argopy.DataFetcher().profile(6904241, 12)
...:
```

- **with the `ds` argument in the data fetcher**:

```
In [5]: argopy.DataFetcher(ds='phy').profile(6902746, 34)
Out[5]:
<datafetcher.erddap>
Name: Ifremer erddap Argo data fetcher for profiles
API: https://erddap.ifremer.fr/erddap
Domain: WMO6902746_CYC34
Performances: cache=False, parallel=False
User mode: standard
Dataset: phy
```

Note: In the future, we could consider to add more mission specific keywords for the `dataset` option and `ds` fetcher argument of *DataFetcher*. This could be *deep* for instance. Please raise an gitHub "issue" if you may require such a new feature.

The bgc dataset

Important: At this time, BGC parameters are only available in `expert` *user mode* and with the `erddap` *data source*.

All **argopy** features work with the **phy** dataset. However, they are some specific methods dedicated to the **bgc** dataset that we now describe.

Specifics in `DataFetcher`

The BGC-Argo Mission gathers data from floats that measure temperature, salinity, pressure and oxygen, pH, nitrate, chlorophyll, backscatter, irradiance down to 2000m. However, beyond this short BGC parameter list there exist in the Argo dataset **more than 120 BGC-related variables**. Therefore, in the *DataFetcher* we implemented 2 specific arguments to handle BGC variables: `params` and `measured`.

The `params` argument

With a *DataFetcher*, the `params` argument can be used to specify which variables will be returned, *whatever their values or availability in BGC floats found in the data selection*.

By default, the `params` argument is set to the keyword `all` to return *all* variables found in the data selection. But the `params` argument can also be a single variable or a list of variables, in which case only these will be returned and all the others discarded.

Syntax example

- To return data from a single BGC parameter, just add it as a string, for instance `DOXY`
- To return more than one BGC parameter, give them as a list of strings, for instance `['DOXY', 'BBP700']`
- To retrieve all available BGC parameters, you can omit the `params` argument (since this is the default value), or give it explicitly as `all`.

```
In [6]: import argopy

In [7]: with argopy.set_options(dataset='bgc', src='erddap', mode='expert'):
   ...:     params = 'all'  # eg: 'DOXY' or ['DOXY', 'BBP700']
   ...:     f = argopy.DataFetcher(params=params)
   ...:     f = f.region([-75, -45, 20, 30, 0, 10, '2021-01', '2021-06'])
   ...:     f.load()
   ...:

In [8]: print(f.data.argo, "\n")  # Easy print of N profiles and points
<xarray.Dataset.argo>
This is a collection of Argo points
N_POINTS(619) ~ N_PROF(46) x N_LEVELS(26)


In [9]: print(list(f.data.data_vars))  # List of dataset variables
['BBP532', 'BBP532_ADJUSTED', 'BBP532_ADJUSTED_ERROR', 'BBP532_ADJUSTED_QC', 'BBP532_
↪DATA_MODE', 'BBP532_QC', 'BBP700', 'BBP700_ADJUSTED', 'BBP700_ADJUSTED_ERROR', 'BBP700_
↪ADJUSTED_QC', 'BBP700_DATA_MODE', 'BBP700_QC', 'CHLA', 'CHLA_ADJUSTED', 'CHLA_ADJUSTED_
↪ERROR', 'CHLA_ADJUSTED_QC', 'CHLA_DATA_MODE', 'CHLA_QC', 'CONFIG_MISSION_NUMBER',
↪'CYCLE_NUMBER', 'DIRECTION', 'DOWNWELLING_PAR', 'DOWNWELLING_PAR_ADJUSTED',
↪'DOWNWELLING_PAR_ADJUSTED_ERROR', 'DOWNWELLING_PAR_ADJUSTED_QC', 'DOWNWELLING_PAR_DATA_
↪MODE', 'DOWNWELLING_PAR_QC', 'DOWN_IRRADIANCE380', 'DOWN_IRRADIANCE380_ADJUSTED',
↪'DOWN_IRRADIANCE380_ADJUSTED_ERROR', 'DOWN_IRRADIANCE380_ADJUSTED_QC', 'DOWN_
↪IRRADIANCE380_DATA_MODE', 'DOWN_IRRADIANCE380_QC', 'DOWN_IRRADIANCE412', 'DOWN_
↪IRRADIANCE412_ADJUSTED', 'DOWN_IRRADIANCE412_ADJUSTED_ERROR', 'DOWN_IRRADIANCE412_
↪ADJUSTED_QC', 'DOWN_IRRADIANCE412_DATA_MODE', 'DOWN_IRRADIANCE412_QC', 'DOWN_
↪IRRADIANCE490', 'DOWN_IRRADIANCE490_ADJUSTED', 'DOWN_IRRADIANCE490_ADJUSTED_ERROR',
↪'DOWN_IRRADIANCE490_ADJUSTED_QC', 'DOWN_IRRADIANCE490_DATA_MODE', 'DOWN_IRRADIANCE490_
↪QC', 'DOXY', 'DOXY_ADJUSTED', 'DOXY_ADJUSTED_ERROR', 'DOXY_ADJUSTED_QC', 'DOXY_DATA_
↪MODE', 'DOXY_QC', 'NITRATE', 'NITRATE_ADJUSTED', 'NITRATE_ADJUSTED_ERROR', 'NITRATE_
↪ADJUSTED_QC', 'NITRATE_DATA_MODE', 'NITRATE_QC', 'PH_IN_SITU_TOTAL', 'PH_IN_SITU_TOTAL_
↪ADJUSTED', 'PH_IN_SITU_TOTAL_ADJUSTED_ERROR', 'PH_IN_SITU_TOTAL_ADJUSTED_QC', 'PH_IN_
↪SITU_TOTAL_DATA_MODE', 'PH_IN_SITU_TOTAL_QC', 'PLATFORM_NUMBER', 'POSITION_QC', 'PRES',
↪ 'PRES_ADJUSTED', 'PRES_ADJUSTED_ERROR', 'PRES_ADJUSTED_QC', 'PRES_DATA_MODE', 'PRES_QC
↪', 'PSAL', 'PSAL_ADJUSTED', 'PSAL_ADJUSTED_ERROR', 'PSAL_ADJUSTED_QC', 'PSAL_DATA_MODE
↪', 'PSAL_QC', 'TEMP', 'TEMP_ADJUSTED', 'TEMP_ADJUSTED_ERROR', 'TEMP_ADJUSTED_QC',
↪'TEMP_DATA_MODE', 'TEMP_QC', 'TIME_QC']
```

The measured argument

With a *DataFetcher*, the measured argument can be used to specify which variables cannot be NaN and must return values. This is very useful to reduce a dataset to points where all or some variables are available.

By default, the measured argument is set to None for unconstrained parameter values. To the opposite, the keyword all requires that all variables found in the data selection cannot be NaNs. In between, you can specific one or more parameters to limit the constrain to a few variables.

Syntax example

Let's impose that some variables cannot be NaNs, for instance DOXY and BBP700:

```
In [10]: import argopy

In [11]: with argopy.set_options(dataset='bgc', src='erddap', mode='expert'):
   ....:     f = argopy.DataFetcher(params='all', measured=['DOXY', 'BBP700'])
   ....:     f = f.region([-75, -45, 20, 30, 0, 10, '2021-01', '2021-06'])
   ....:     f.load()
   ....:

In [12]: print(f.data.argo, "\n")  # Easy print of N profiles and points
<xarray.Dataset.argo>
This is a collection of Argo points
N_POINTS(45) ~ N_PROF(45) x N_LEVELS(1)


In [13]: print(list(f.data.data_vars))  # List of dataset variables
['BBP532', 'BBP532_ADJUSTED', 'BBP532_ADJUSTED_ERROR', 'BBP532_ADJUSTED_QC', 'BBP532_
↪DATA_MODE', 'BBP532_QC', 'BBP700', 'BBP700_ADJUSTED', 'BBP700_ADJUSTED_ERROR', 'BBP700_
↪ADJUSTED_QC', 'BBP700_DATA_MODE', 'BBP700_QC', 'CHLA', 'CHLA_ADJUSTED', 'CHLA_ADJUSTED_
↪ERROR', 'CHLA_ADJUSTED_QC', 'CHLA_DATA_MODE', 'CHLA_QC', 'CONFIG_MISSION_NUMBER',
↪'CYCLE_NUMBER', 'DIRECTION', 'DOWNWELLING_PAR', 'DOWNWELLING_PAR_ADJUSTED',
↪'DOWNWELLING_PAR_ADJUSTED_ERROR', 'DOWNWELLING_PAR_ADJUSTED_QC', 'DOWNWELLING_PAR_DATA_
↪MODE', 'DOWNWELLING_PAR_QC', 'DOWN_IRRADIANCE380', 'DOWN_IRRADIANCE380_ADJUSTED',
↪'DOWN_IRRADIANCE380_ADJUSTED_ERROR', 'DOWN_IRRADIANCE380_ADJUSTED_QC', 'DOWN_
↪IRRADIANCE380_DATA_MODE', 'DOWN_IRRADIANCE380_QC', 'DOWN_IRRADIANCE412', 'DOWN_
↪IRRADIANCE412_ADJUSTED', 'DOWN_IRRADIANCE412_ADJUSTED_ERROR', 'DOWN_IRRADIANCE412_
↪ADJUSTED_QC', 'DOWN_IRRADIANCE412_DATA_MODE', 'DOWN_IRRADIANCE412_QC', 'DOWN_
↪IRRADIANCE490', 'DOWN_IRRADIANCE490_ADJUSTED', 'DOWN_IRRADIANCE490_ADJUSTED_ERROR',
↪'DOWN_IRRADIANCE490_ADJUSTED_QC', 'DOWN_IRRADIANCE490_DATA_MODE', 'DOWN_IRRADIANCE490_
↪QC', 'DOXY', 'DOXY_ADJUSTED', 'DOXY_ADJUSTED_ERROR', 'DOXY_ADJUSTED_QC', 'DOXY_DATA_
↪MODE', 'DOXY_QC', 'NITRATE', 'NITRATE_ADJUSTED', 'NITRATE_ADJUSTED_ERROR', 'NITRATE_
↪ADJUSTED_QC', 'NITRATE_DATA_MODE', 'NITRATE_QC', 'PH_IN_SITU_TOTAL', 'PH_IN_SITU_TOTAL_
↪ADJUSTED', 'PH_IN_SITU_TOTAL_ADJUSTED_ERROR', 'PH_IN_SITU_TOTAL_ADJUSTED_QC', 'PH_IN_
↪SITU_TOTAL_DATA_MODE', 'PH_IN_SITU_TOTAL_QC', 'PLATFORM_NUMBER', 'POSITION_QC', 'PRES',
↪ 'PRES_ADJUSTED', 'PRES_ADJUSTED_ERROR', 'PRES_ADJUSTED_QC', 'PRES_DATA_MODE', 'PRES_QC
↪', 'PSAL', 'PSAL_ADJUSTED', 'PSAL_ADJUSTED_ERROR', 'PSAL_ADJUSTED_QC', 'PSAL_DATA_MODE
↪', 'PSAL_QC', 'TEMP', 'TEMP_ADJUSTED', 'TEMP_ADJUSTED_ERROR', 'TEMP_ADJUSTED_QC',
↪'TEMP_DATA_MODE', 'TEMP_QC', 'TIME_QC']
```

We can see from f.data.argo.N_POINTS that the dataset is reduced compared to the previous version without constraints on variables **measured**.

Specifics in `ArgoIndex`

All details and examples of the BGC specifics methods for `ArgoIndex` can be found in: *Usage with bgc index*.

1.6.4 User mode (, ,)

Hint: **argopy** manipulates the raw data to make them easier to work with. To make sure you understand the data you're getting, have a look to this section.

Contents

Problem

For non-experts of the Argo dataset, it can be quite complicated to get access to Argo measurements. Indeed, the Argo data set is very complex, with thousands of different variables, tens of reference tables and a user manual more than 100 pages long.

This is mainly due to:

- Argo measurements coming from many different models of floats or sensors,
- quality control of *in situ* measurements of autonomous platforms being really a matter of ocean and data experts,
- the Argo data management workflow being distributed between more than 10 Data Assembly Centers all around the world,
- the Argo autonomous profiling floats, despite quite a simple principle of functioning, is a rather complex robot that needs a lot of data to be monitored and logged.

Solution

In order to ease Argo data analysis for the vast majority of users, we implemented in **argopy** different levels of verbosity and data processing to hide or simply remove variables only meaningful to experts.

User mode details

argopy provides 3 user modes:

- **expert** mode return all the Argo data, without any postprocessing,
- **standard** mode simplifies the dataset, remove most of its jargon and return *a priori* good data,
- **research** mode simplifies the dataset to its heart, preserving only data of the highest quality for research studies, including studies sensitive to small pressure and salinity bias (e.g. calculations of global ocean heat content or mixed layer depth).

In **standard** and **research** modes, fetched data are automatically filtered to account for their quality (using the *quality control flags*) and level of processing by the data centers (considering for each parameter the data mode which indicates if a human expert has carefully looked at the data or not). Both mode return a postprocessed subset of the full Argo dataset.

Hence the main difference between the **standard** and **research** modes is in the level of data quality insurance. In **standard** mode, only good or probably good data are returned and includes real time data that have been validated automatically but not by a human expert. The **research** mode is the safer choice, with data of the highest quality, carefully checked in delayed mode by a human expert of the Argo Data Management Team.

Table 2: Table of **argopy** user mode data processing details for **physical** parameters (phy dataset)

	expert	standard	research
Level of quality (QC flags) retained	all	good or probably good (QC=[1,2])	good (QC=1)
Level of assessment (Data mode) retained	all: [R,D,A] modes	all: [R,D,A] modes, but PARAM_ADJUSTED and PARAM are merged in a single variable according to the mode	best only (D mode only)
Pressure error	any	any	smaller than 20db
Variables returned	all	all without jargon (DATA_MODE and QC_FLAG are retained)	comprehensive minimum

About the bgc dataset

The table of **argopy** user mode data processing details for **biogeochemical** parameters is being defined (#280) and will be implemented in a near future release.

How to select a user mode ?

Let's import the **argopy** data fetcher:

```
In [1]: import argopy

In [2]: from argopy import DataFetcher as ArgoDataFetcher
```

By default, all **argopy** data fetchers are set to work with a **standard** user mode.

If you want to change the user mode, or to simply makes it explicit in your code, you can use one of the following 3 methods:

- the **argopy** global option setter:

```
In [3]: argopy.set_options(mode='standard')
Out[3]: <argopy.options.set_options at 0x7fd6f86398b0>
```

- a temporary **context**:

```
In [4]: with argopy.set_options(mode='expert'):
   ...:     ArgoDataFetcher().profile(6902746, 34)
   ...:
```

- or the **fetcher option**:

```
In [5]: ArgoDataFetcher(mode='research').profile(6902746, 34)
Out[5]:
<datafetcher.erddap>
Name: Ifremer erddap Argo data fetcher for profiles
API: https://erddap.ifremer.fr/erddap
Domain: WMO6902746_CYC34
Performances: cache=False, parallel=False
User mode: research
Dataset: phy
```

Example of differences in user modes

To highlight differences in data returned for each user modes, let's compare data fetched for one profile.

You will note that the **standard** and **research** modes have fewer variables to let you focus on your analysis. For **expert**, all Argo variables for you to work with are here.

In **expert** mode:

In **standard** mode:

In **research** mode:

```
In [6]: with argopy.set_options(mode='expert'):
   ...:     ds = ArgoDataFetcher(src='gdac').profile(6902755, 12).to_xarray()
   ...:     print(ds.data_vars)
   ...:
Data variables:
    CONFIG_MISSION_NUMBER     (N_POINTS) int64 2 2 2 2 2 2 2 2 ... 2 2 2 2 2 2 2 2
    CYCLE_NUMBER              (N_POINTS) int64 12 12 12 12 12 ... 12 12 12 12 12
    DATA_CENTRE               (N_POINTS) <U2 'IF' 'IF' 'IF' ... 'IF' 'IF' 'IF'
    DATA_MODE                 (N_POINTS) <U1 'D' 'D' 'D' 'D' ... 'D' 'R' 'R' 'R'
    DATA_STATE_INDICATOR      (N_POINTS) <U4 '2C  ' '2C  ' ... '2B  ' '2B  '
    DC_REFERENCE              (N_POINTS) <U32 '                                ...
    DIRECTION                 (N_POINTS) <U1 'A' 'A' 'A' 'A' ... 'A' 'A' 'A' 'A'
    FIRMWARE_VERSION          (N_POINTS) <U32 '5611A02                         ...
    FLOAT_SERIAL_NO           (N_POINTS) <U32 'AR2000-16FR012                  ...
    PI_NAME                   (N_POINTS) <U64 'Virginie THIERRY                ...
    PLATFORM_NUMBER           (N_POINTS) int64 6902755 6902755 ... 6902755
    PLATFORM_TYPE             (N_POINTS) <U32 'ARVOR                           ...
    POSITIONING_SYSTEM        (N_POINTS) <U8 'ARGOS   ' ... 'ARGOS   '
    POSITION_QC               (N_POINTS) int64 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1
    PRES                      (N_POINTS) float32 3.0 4.0 5.0 6.0 ... 0.0 1.0 2.0
    PRES_ADJUSTED             (N_POINTS) float32 3.0 4.0 5.0 6.0 ... nan nan nan
    PRES_ADJUSTED_ERROR       (N_POINTS) float32 2.4 2.4 2.4 2.4 ... nan nan nan
    PRES_ADJUSTED_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 ... 1 1 1 1 0 0 0 0
    PRES_QC                   (N_POINTS) int64 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1
    PROFILE_PRES_QC           (N_POINTS) <U1 'A' 'A' 'A' 'A' ... 'A' 'A' 'A' 'A'
    PROFILE_PSAL_QC           (N_POINTS) <U1 'A' 'A' 'A' 'A' ... 'A' 'F' 'F' 'F'
    PROFILE_TEMP_QC           (N_POINTS) <U1 'A' 'A' 'A' 'A' ... 'A' 'D' 'D' 'D'
    PROJECT_NAME              (N_POINTS) <U64 'ASFAR                           ...
    PSAL                      (N_POINTS) float32 34.87 34.87 ... 34.87 34.87
    PSAL_ADJUSTED             (N_POINTS) float32 34.87 34.87 34.87 ... nan nan
    PSAL_ADJUSTED_ERROR       (N_POINTS) float32 0.01 0.01 0.01 ... nan nan nan
```

(continues on next page)

(continued from previous page)

```
    PSAL_ADJUSTED_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 ... 1 1 1 1 0 0 0
    PSAL_QC                   (N_POINTS) int64 1 1 1 1 1 1 1 1 ... 1 1 1 1 3 3 3
    TEMP                      (N_POINTS) float32 7.598 7.599 ... 7.598 7.599
    TEMP_ADJUSTED             (N_POINTS) float32 7.598 7.599 7.602 ... nan nan
    TEMP_ADJUSTED_ERROR       (N_POINTS) float32 0.002 0.002 0.002 ... nan nan
    TEMP_ADJUSTED_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 ... 1 1 1 1 0 0 0
    TEMP_QC                   (N_POINTS) int64 1 1 1 1 1 1 1 1 ... 1 1 1 1 3 3 1
    TIME_LOCATION             (N_POINTS) datetime64[ns] 2018-10-20T00:17:05.0...
    TIME_QC                   (N_POINTS) int64 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1
    VERTICAL_SAMPLING_SCHEME  (N_POINTS) <U256 'Primary sampling: averaged [1...
    WMO_INST_TYPE             (N_POINTS) int64 844 844 844 844 ... 844 844 844
```

```
In [7]: with argopy.set_options(mode='standard'):
   ...:     ds = ArgoDataFetcher(src='gdac').profile(6902755, 12).to_xarray()
   ...:     print(ds.data_vars)
   ...:
Data variables:
    CYCLE_NUMBER     (N_POINTS) int64 12 12 12 12 12 12 12 ... 12 12 12 12 12 12
    DATA_MODE        (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D' 'D'
    DIRECTION        (N_POINTS) <U1 'A' 'A' 'A' 'A' 'A' ... 'A' 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER  (N_POINTS) int64 6902755 6902755 ... 6902755 6902755
    POSITION_QC      (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    PRES             (N_POINTS) float64 3.0 4.0 5.0 ... 1.713e+03 1.732e+03
    PRES_ERROR       (N_POINTS) float64 2.4 2.4 2.4 2.4 2.4 ... 2.4 2.4 2.4 2.4
    PRES_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    PSAL             (N_POINTS) float64 34.87 34.87 34.87 ... 34.94 34.94 34.94
    PSAL_ERROR       (N_POINTS) float64 0.01 0.01 0.01 0.01 ... 0.01 0.01 0.01
    PSAL_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TEMP             (N_POINTS) float64 7.598 7.599 7.602 ... 3.588 3.549 3.536
    TEMP_ERROR       (N_POINTS) float64 0.002 0.002 0.002 ... 0.002 0.002 0.002
    TEMP_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TIME_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
```

```
In [8]: with argopy.set_options(mode='research'):
   ...:     ds = ArgoDataFetcher(src='gdac').profile(6902755, 12).to_xarray()
   ...:     print(ds.data_vars)
   ...:
Data variables:
    CYCLE_NUMBER     (N_POINTS) int64 12 12 12 12 12 12 12 ... 12 12 12 12 12 12
    DIRECTION        (N_POINTS) <U1 'A' 'A' 'A' 'A' 'A' ... 'A' 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER  (N_POINTS) int64 6902755 6902755 ... 6902755 6902755
    PRES             (N_POINTS) float64 3.0 4.0 5.0 ... 1.713e+03 1.732e+03
    PRES_ERROR       (N_POINTS) float64 2.4 2.4 2.4 2.4 2.4 ... 2.4 2.4 2.4 2.4
    PSAL             (N_POINTS) float64 34.87 34.87 34.87 ... 34.94 34.94 34.94
    PSAL_ERROR       (N_POINTS) float64 0.01 0.01 0.01 0.01 ... 0.01 0.01 0.01
    TEMP             (N_POINTS) float64 7.598 7.599 7.602 ... 3.588 3.549 3.536
    TEMP_ERROR       (N_POINTS) float64 0.002 0.002 0.002 ... 0.002 0.002 0.002
```

Note: A note for **expert** users looking at **standard** and **research** mode results: they are no `PARAM_ADJUSTED` variables because they've been renamed `PARAM` wherever the `DATA_MODE` variable was `ADJUSTED` or `DELAYED`.

1.6.5 In a nutshell

2 lines to download Argo data: import and fetch !

```
In [1]: import argopy

In [2]: ds = argopy.DataFetcher().region([-75, -45, 20, 30, 0, 10, '2011-01', '2011-06
↪']).load().data
```

```
In [3]: ds
Out[3]:
<xarray.Dataset>
Dimensions:          (N_POINTS: 998)
Coordinates:
  * N_POINTS         (N_POINTS) int64 0 1 2 3 4 5 6 ... 992 993 994 995 996 997
    LATITUDE         (N_POINTS) float64 24.54 24.54 25.04 ... 26.67 24.96 24.96
    LONGITUDE        (N_POINTS) float64 -45.14 -45.14 -51.58 ... -50.4 -50.4
    TIME             (N_POINTS) datetime64[ns] 2011-01-01T11:49:19 ... 2011-0...
Data variables: (12/15)
    CYCLE_NUMBER     (N_POINTS) int64 23 23 10 10 10 10 10 ... 1 5 2 10 10 38 38
    DATA_MODE        (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D' 'D'
    DIRECTION        (N_POINTS) <U1 'A' 'A' 'A' 'A' 'A' ... 'A' 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER  (N_POINTS) int64 1901463 1901463 ... 1901463 1901463
    POSITION_QC      (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    PRES             (N_POINTS) float64 5.0 10.0 2.0 4.0 ... 5.12 9.42 5.0 10.0
    ...               ...
    PSAL_ERROR       (N_POINTS) float32 0.01 0.01 0.01 ... 0.01 0.01091 0.01182
    PSAL_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TEMP             (N_POINTS) float64 24.08 24.08 24.03 ... 25.64 25.1 24.79
    TEMP_ERROR       (N_POINTS) float32 0.002 0.002 0.002 ... 0.0025 0.002 0.002
    TEMP_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TIME_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
Attributes:
    DATA_ID:              ARGO
    DOI:                  http://doi.org/10.17882/42182
    Fetched_from:         https://erddap.ifremer.fr/erddap
    Fetched_by:           docs
    Fetched_date:         2024/04/22
    Fetched_constraints:  [x=-75.00/-45.00; y=20.00/30.00; z=0.0/10.0; t=2011...
    Fetched_uri:          ['https://erddap.ifremer.fr/erddap/tabledap/ArgoFlo...
    history:              Variables filtered according to DATA_MODE; Variable...
```

1.6.6 Workflow explained

Let's explain what happened in the single line Argo data fetching above.

1 we create a DataFetcher

2 we select data

3 then we fetch data

4 Viz data

Import **argopy** and create a instance of *DataFetcher*:

```
In [4]: import argopy

In [5]: f = argopy.DataFetcher()

In [6]: f
Out[6]:
<datafetcher.erddap> 'No access point initialised'
Available access points: float, profile, region
Performances: cache=False, parallel=False
User mode: standard
Dataset: phy
```

By default, **argopy** will load the `phy` *dataset*, in `standard` *user mode* from the `erddap` *data source*.

Once you have a `DataFetcher`, you must select data. As an example, here is a space/time data selection:

```
In [7]: f = f.region([-75, -45, 20, 30, 0, 10, '2011-01', '2011-06'])

In [8]: f
Out[8]:
<datafetcher.erddap>
Name: Ifremer erddap Argo data fetcher for a space/time region
API: https://erddap.ifremer.fr/erddap
Domain: [x=-75.00/-45.00; y=20.00/30.0 ... 10.0; t=2011-01-01/2011-06-01]
Performances: cache=False, parallel=False
User mode: standard
Dataset: phy
```

See *all data selector methods here*.

Once you defined a data selection, data fetching will be triggered if you access one of the `DataFetcher` properties:

- `data`, this is a `xarray.Dataset` with all Argo data in the selection,
- `index`, this is a `pandas.DataFrame` with a list of profiles in the selection.

```
In [9]: f.data
Out[9]:
<xarray.Dataset>
Dimensions:          (N_POINTS: 998)
Coordinates:
  * N_POINTS         (N_POINTS) int64 0 1 2 3 4 5 6 ... 992 993 994 995 996 997
    LATITUDE         (N_POINTS) float64 24.54 24.54 25.04 ... 26.67 24.96 24.96
    LONGITUDE        (N_POINTS) float64 -45.14 -45.14 -51.58 ... -50.4 -50.4
    TIME             (N_POINTS) datetime64[ns] 2011-01-01T11:49:19 ... 2011-0...
Data variables: (12/15)
    CYCLE_NUMBER     (N_POINTS) int64 23 23 10 10 10 10 10 ... 1 5 2 10 10 38 38
    DATA_MODE        (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D' 'D'
    DIRECTION        (N_POINTS) <U1 'A' 'A' 'A' 'A' 'A' ... 'A' 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER  (N_POINTS) int64 1901463 1901463 ... 1901463 1901463
    POSITION_QC      (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    PRES             (N_POINTS) float64 5.0 10.0 2.0 4.0 ... 5.12 9.42 5.0 10.0
    ...               ...
    PSAL_ERROR       (N_POINTS) float32 0.01 0.01 0.01 ... 0.01 0.01091 0.01182
    PSAL_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
```

(continues on next page)

(continued from previous page)

```
    TEMP             (N_POINTS) float64 24.08 24.08 24.03 ... 25.64 25.1 24.79
    TEMP_ERROR       (N_POINTS) float32 0.002 0.002 0.002 ... 0.0025 0.002 0.002
    TEMP_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1
    TIME_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1
Attributes:
    DATA_ID:              ARGO
    DOI:                  http://doi.org/10.17882/42182
    Fetched_from:         https://erddap.ifremer.fr/erddap
    Fetched_by:           docs
    Fetched_date:         2024/04/22
    Fetched_constraints:  [x=-75.00/-45.00; y=20.00/30.00; z=0.0/10.0; t=2011...
    Fetched_uri:          ['https://erddap.ifremer.fr/erddap/tabledap/ArgoFlo...
    history:              Variables filtered according to DATA_MODE; Variable...
```

```
In [10]: f.index
Out[10]:
                   date  latitude  longitude      wmo  cyc
0   2011-01-02 11:26:40    27.829    -56.303  1901461   23
1   2011-01-12 11:44:20    27.988    -56.378  1901461   24
2   2011-01-22 11:23:26    27.891    -55.865  1901461   25
3   2011-02-01 11:18:38    27.538    -54.669  1901461   26
4   2011-02-11 11:46:04    27.487    -53.686  1901461   27
..                  ...       ...        ...      ...  ...
462 2011-05-02 01:22:10    27.134    -71.040  6901050    2
463 2011-05-12 01:25:11    27.498    -70.044  6901050    3
464 2011-05-20 16:30:09    21.120    -57.985  6901051    1
465 2011-05-30 16:18:21    21.489    -58.071  6901051    2
466 2011-05-25 20:08:09    21.072    -52.625  6901052    1

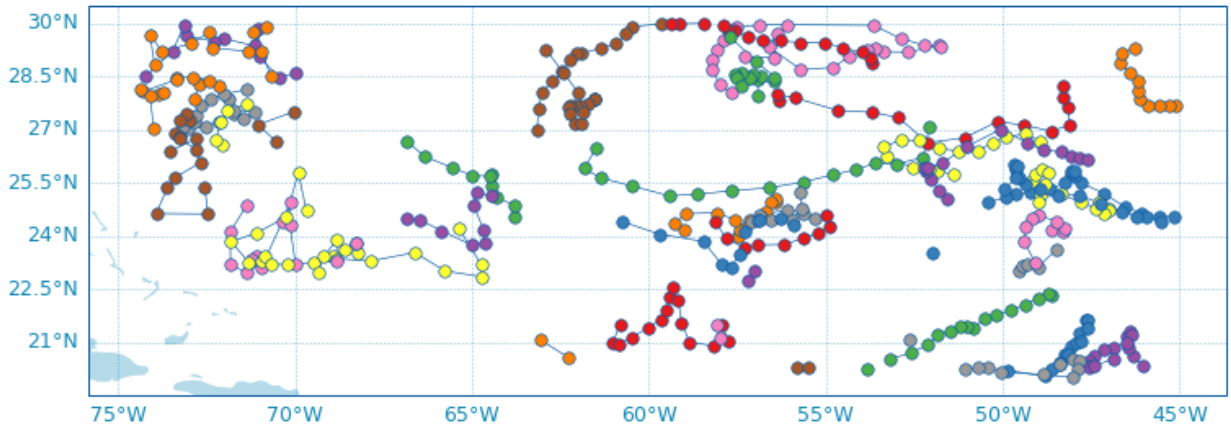
[467 rows x 5 columns]
```

These fetcher properties call on the *DataFetcher.load()* method, which in turn, calls either *DataFetcher.to_xarray()* or *DataFetcher.to_index()* if data structures are not already in memory.

Note that the *DataFetcher.to_xarray()* and *DataFetcher.to_index()* will force data download on every call, while the *DataFetcher.load()* method will keep data in memory in the *DataFetcher.data* and *DataFetcher.index* properties.

If you wish to quickly look at your data selection, you can call on the *DataFetcher.plot()*.

```
f.plot('trajectory', add_legend=False)
```



If you selected data for a float, the *DataFetcher.dashboard()* method can also be used.

See the *Data visualisation* section for more details on **argopy** data visualisation tools.

Hint: The *DataFetcher.domain* property will also give you the space/time domain covered by your data selection.

```
In [11]: f.domain  # [lon_min, lon_max, lat_min, lat_max, pres_min, pres_max, date_min,␣
↪date_max]
Out[11]:
[-74.37700000000001,
 -45.118,
 20.018,
 29.995,
 1.0,
 10.399999618530273,
 numpy.datetime64('2011-01-01T11:49:19.000000000'),
 numpy.datetime64('2011-05-31T11:34:52.000000000')]
```

1.7 Working with Argo data

argopy not only get you easy access to Argo data, it also aims to help you work with it.

In the following documentation sections, you will see how to:

- *manipulate* a `xarray.Dataset` with the `argo` accessor methods,
- *visualize* Argo data, weather it is a `xarray.Dataset` or `pandas.DataFrame` profile index,
- use **argopy** helper methods for *Argo quality control*.

1.7.1 Manipulating data

Once you fetched data, **argopy** comes with a handy `xarray.Dataset` accessor `argo` to perform specific manipulation of the data. This means that if your dataset is named *ds*, then you can use *ds.argo* to access more **argopy** functions. The full list is available in the API documentation page *Dataset.argo (xarray accessor)*.

Let's start with standard import:

```
In [1]: from argopy import DataFetcher
```

Transformation

Points vs profiles

By default, fetched data are returned as a 1D array collection of measurements:

```
In [2]: f = DataFetcher().region([-75,-55,30.,40.,0,100., '2011-01-01', '2011-01-15'])

In [3]: ds_points = f.data

In [4]: ds_points
Out[4]:
<xarray.Dataset>
Dimensions:          (N_POINTS: 524)
Coordinates:
  * N_POINTS         (N_POINTS) int64 0 1 2 3 4 5 6 ... 518 519 520 521 522 523
    LATITUDE         (N_POINTS) float64 37.28 37.28 37.28 ... 33.07 33.07 33.07
    LONGITUDE        (N_POINTS) float64 -66.77 -66.77 -66.77 ... -64.59 -64.59
    TIME             (N_POINTS) datetime64[ns] 2011-01-02T11:14:06 ... 2011-0...
Data variables: (12/15)
    CYCLE_NUMBER     (N_POINTS) int64 150 150 150 150 150 150 ... 13 13 13 13 13
    DATA_MODE        (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D' 'D'
    DIRECTION        (N_POINTS) <U1 'A' 'A' 'A' 'A' 'A' ... 'A' 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER  (N_POINTS) int64 4900803 4900803 ... 5903377 5903377
    POSITION_QC      (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    PRES             (N_POINTS) float64 5.0 10.0 15.0 20.0 ... 95.97 97.97 99.97
    ...               ...
    PSAL_ERROR       (N_POINTS) float64 0.01 0.01 0.01 0.01 ... 0.01 0.01 0.01
    PSAL_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
```

(continues on next page)

(continued from previous page)

```
    TEMP             (N_POINTS) float64 19.46 19.47 19.47 ... 19.2 19.2 19.2
    TEMP_ERROR       (N_POINTS) float64 0.002 0.002 0.002 ... 0.002 0.002 0.002
    TEMP_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TIME_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
Attributes:
    DATA_ID:              ARGO
    DOI:                  http://doi.org/10.17882/42182
    Fetched_from:         https://erddap.ifremer.fr/erddap
    Fetched_by:           docs
    Fetched_date:         2024/04/22
    Fetched_constraints:  [x=-75.00/-55.00; y=30.00/40.00; z=0.0/100.0; t=201...
    Fetched_uri:          ['https://erddap.ifremer.fr/erddap/tabledap/ArgoFlo...
    history:              Variables filtered according to DATA_MODE; Variable...
```

If you prefer to work with a 2D array collection of vertical profiles, simply transform the dataset with *Dataset.argo.point2profile()*:

```
In [5]: ds_profiles = ds_points.argo.point2profile()

In [6]: ds_profiles
Out[6]:
<xarray.Dataset>
Dimensions:           (N_PROF: 18, N_LEVELS: 50)
Coordinates:
  * N_PROF            (N_PROF) int64 7 13 15 0 6 2 9 4 11 5 1 12 10 17 3 8 14 16
  * N_LEVELS          (N_LEVELS) int64 0 1 2 3 4 5 6 7 ... 43 44 45 46 47 48 49
    LATITUDE          (N_PROF) float64 37.28 33.98 32.88 ... 37.03 34.39 33.07
    LONGITUDE         (N_PROF) float64 -66.77 -71.17 -64.93 ... -72.75 -64.59
    TIME              (N_PROF) datetime64[ns] 2011-01-02T11:14:06 ... 2011-01-...
Data variables: (12/15)
    CYCLE_NUMBER      (N_PROF) int64 150 3 11 100 180 280 ... 17 62 148 151 4 13
    DATA_MODE         (N_PROF) <U1 'D' 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D' 'D'
    DIRECTION         (N_PROF) <U1 'A' 'A' 'A' 'A' 'A' ... 'A' 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER   (N_PROF) int64 4900803 4901218 5903377 ... 4901218 5903377
    POSITION_QC       (N_PROF) int64 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
    PRES              (N_PROF, N_LEVELS) float64 5.0 10.0 15.0 ... 99.97 nan
    ...                ...
    PSAL_ERROR        (N_PROF, N_LEVELS) float64 0.01 0.01 0.01 ... 0.01 0.01 nan
    PSAL_QC           (N_PROF) int64 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
    TEMP              (N_PROF, N_LEVELS) float64 19.46 19.47 19.47 ... 19.2 nan
    TEMP_ERROR        (N_PROF) float64 0.002 0.002 0.002 ... 0.002 0.002 0.002
    TEMP_QC           (N_PROF) int64 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
    TIME_QC           (N_PROF) int64 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Attributes:
    DATA_ID:              ARGO
    DOI:                  http://doi.org/10.17882/42182
    Fetched_from:         https://erddap.ifremer.fr/erddap
    Fetched_by:           docs
    Fetched_date:         2024/04/22
    Fetched_constraints:  [x=-75.00/-55.00; y=30.00/40.00; z=0.0/100.0; t=201...
    Fetched_uri:          ['https://erddap.ifremer.fr/erddap/tabledap/ArgoFlo...
    history:              Variables filtered according to DATA_MODE; Variable...
```

You can simply reverse this transformation with the *Dataset.argo.profile2point()*:

```
In [7]: ds = ds_profiles.argo.profile2point()

In [8]: ds
Out[8]:
<xarray.Dataset>
Dimensions:          (N_POINTS: 524)
Coordinates:
    LONGITUDE        (N_POINTS) float64 -66.77 -66.77 -66.77 ... -64.59 -64.59
    TIME             (N_POINTS) datetime64[ns] 2011-01-02T11:14:06 ... 2011-0...
    LATITUDE         (N_POINTS) float64 37.28 37.28 37.28 ... 33.07 33.07 33.07
  * N_POINTS         (N_POINTS) int64 0 1 2 3 4 5 6 ... 518 519 520 521 522 523
Data variables: (12/15)
    CYCLE_NUMBER     (N_POINTS) int64 150 150 150 150 150 150 ... 13 13 13 13 13
    DATA_MODE        (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D' 'D'
    DIRECTION        (N_POINTS) <U1 'A' 'A' 'A' 'A' 'A' ... 'A' 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER  (N_POINTS) int64 4900803 4900803 ... 5903377 5903377
    POSITION_QC      (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    PRES             (N_POINTS) float64 5.0 10.0 15.0 20.0 ... 95.97 97.97 99.97
    ...               ...
    PSAL_ERROR       (N_POINTS) float64 0.01 0.01 0.01 0.01 ... 0.01 0.01 0.01
    PSAL_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TEMP             (N_POINTS) float64 19.46 19.47 19.47 ... 19.2 19.2 19.2
    TEMP_ERROR       (N_POINTS) float64 0.002 0.002 0.002 ... 0.002 0.002 0.002
    TEMP_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TIME_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
Attributes:
    DATA_ID:              ARGO
    DOI:                  http://doi.org/10.17882/42182
    Fetched_from:         https://erddap.ifremer.fr/erddap
    Fetched_by:           docs
    Fetched_date:         2024/04/22
    Fetched_constraints:  [x=-75.00/-55.00; y=30.00/40.00; z=0.0/100.0; t=201...
    Fetched_uri:          ['https://erddap.ifremer.fr/erddap/tabledap/ArgoFlo...
    history:              Variables filtered according to DATA_MODE; Variable...
```

Pressure levels: Interpolation

Once your dataset is a collection of vertical **profiles**, you can interpolate variables on standard pressure levels using *Dataset.argo.interp_std_levels()* with your levels as input:

```
In [9]: ds_interp = ds_profiles.argo.interp_std_levels([0,10,20,30,40,50])

In [10]: ds_interp
Out[10]:
<xarray.Dataset>
Dimensions:            (N_PROF: 18, PRES_INTERPOLATED: 6)
Coordinates:
  * N_PROF             (N_PROF) int64 7 13 15 0 6 2 9 4 ... 1 12 10 17 3 8 14 16
    LATITUDE           (N_PROF) float64 37.28 33.98 32.88 ... 37.03 34.39 33.07
    LONGITUDE          (N_PROF) float64 -66.77 -71.17 -64.93 ... -72.75 -64.59
```

(continues on next page)

(continued from previous page)

```
    TIME                (N_PROF) datetime64[ns] 2011-01-02T11:14:06 ... 2011-0...
  * PRES_INTERPOLATED   (PRES_INTERPOLATED) int64 0 10 20 30 40 50
Data variables:
    CYCLE_NUMBER        (N_PROF) int64 150 3 11 100 180 280 ... 62 148 151 4 13
    DATA_MODE           (N_PROF) <U1 'D' 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D' 'D'
    DIRECTION           (N_PROF) <U1 'A' 'A' 'A' 'A' 'A' ... 'A' 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER     (N_PROF) int64 4900803 4901218 ... 4901218 5903377
    PRES                (N_PROF, PRES_INTERPOLATED) float64 5.0 10.0 ... 50.0
    PSAL                (N_PROF, PRES_INTERPOLATED) float64 36.67 36.67 ... 36.68
    TEMP                (N_PROF, PRES_INTERPOLATED) float64 19.46 19.47 ... 19.24
Attributes:
    DATA_ID:              ARGO
    DOI:                  http://doi.org/10.17882/42182
    Fetched_from:         https://erddap.ifremer.fr/erddap
    Fetched_by:           docs
    Fetched_date:         2024/04/22
    Fetched_constraints:  [x=-75.00/-55.00; y=30.00/40.00; z=0.0/100.0; t=201...
    Fetched_uri:          ['https://erddap.ifremer.fr/erddap/tabledap/ArgoFlo...
    history:              Variables filtered according to DATA_MODE; Variable...
```

Note on the linear interpolation process :

- Only profiles that have a maximum pressure higher than the highest standard level are selected for interpolation.
- Remaining profiles must have at least five data points to allow interpolation.
- For each profile, shallowest data point is repeated to the surface to allow a 0 standard level while avoiding extrapolation.

Pressure levels: Group-by bins

If you prefer to avoid interpolation, you can opt for a pressure bins grouping reduction using *Dataset.argo.groupby_pressure_bins()*. This method can be used to subsample and align an irregular dataset (pressure not being similar in all profiles) on a set of pressure bins. The output dataset could then be used to perform statistics along the N_PROF dimension because N_LEVELS will corresponds to similar pressure bins.

To illustrate this method, let's start by fetching some data from a low vertical resolution float:

```
In [11]: f = DataFetcher(src='erddap', mode='expert').float(2901623)  # Low res float

In [12]: ds = f.data
```

Let's now sub-sample these measurements along 250db bins, selecting values from the **deepest** pressure levels for each bins:

```
In [13]: bins = np.arange(0.0, np.max(ds["PRES"]), 250.0)

In [14]: ds_binned = ds.argo.groupby_pressure_bins(bins=bins, select='deep')

In [15]: ds_binned
Out[15]:
<xarray.Dataset>
```

(continues on next page)

(continued from previous page)

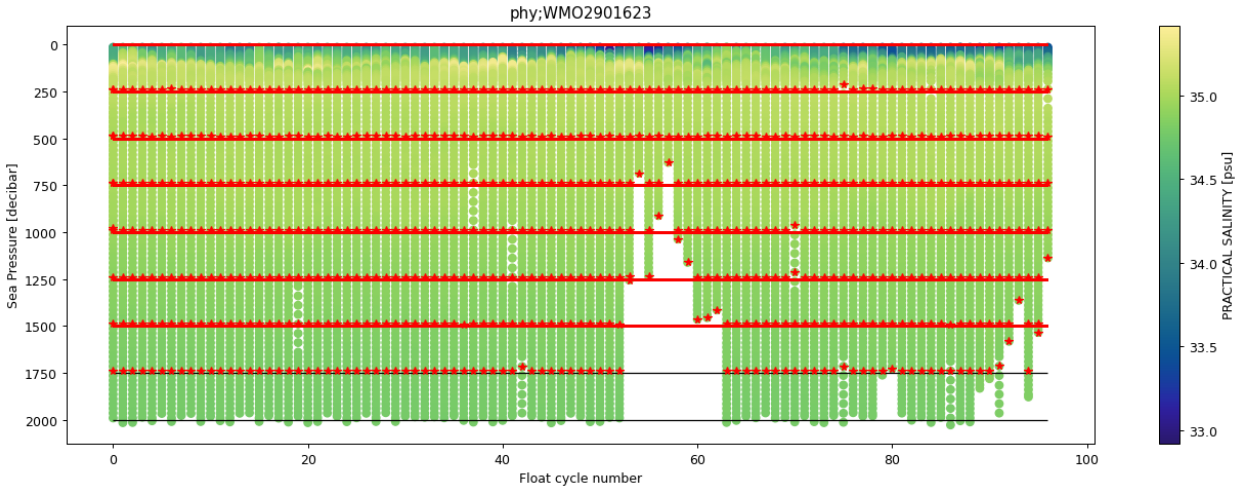
```
Dimensions:                   (N_POINTS: 659)
Coordinates:
    LONGITUDE                 (N_POINTS) float64 92.28 92.28 ... 94.77 94.77
    TIME                      (N_POINTS) datetime64[ns] 2010-05-14T03:35:00 ....
    LATITUDE                  (N_POINTS) float64 0.012 0.012 ... 3.388 3.388
    STD_PRES_BINS             (N_POINTS) float64 0.0 250.0 500.0 ... 750.0 1e+03
  * N_POINTS                  (N_POINTS) int64 0 1 2 3 4 ... 654 655 656 657 658
Data variables: (12/23)
    CONFIG_MISSION_NUMBER     (N_POINTS) int64 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1
    CYCLE_NUMBER              (N_POINTS) int64 0 0 0 0 0 0 ... 95 96 96 96 96 96
    DATA_MODE                 (N_POINTS) <U1 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D'
    DIRECTION                 (N_POINTS) <U1 'D' 'D' 'D' 'D' ... 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER           (N_POINTS) int64 2901623 2901623 ... 2901623
    POSITION_QC               (N_POINTS) int64 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1
    ...                        ...
    TEMP_ADJUSTED             (N_POINTS) float32 13.17 10.08 ... 6.551 6.071
    TEMP_ADJUSTED_ERROR       (N_POINTS) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0
    TEMP_ADJUSTED_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1
    TEMP_QC                   (N_POINTS) int64 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1
    TIME_QC                   (N_POINTS) int64 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1
    VERTICAL_SAMPLING_SCHEME  (N_POINTS) <U29 'Primary sampling: discrete []'...
Attributes:
    DATA_ID:              ARGO
    DOI:                  http://doi.org/10.17882/42182
    Fetched_from:         https://erddap.ifremer.fr/erddap
    Fetched_by:           docs
    Fetched_date:         2024/04/22
    Fetched_constraints:  WMO2901623
    Fetched_uri:          ['https://erddap.ifremer.fr/erddap/tabledap/ArgoFlo...
    history:              Transformed with point2profile; Sub-sampled and re-...
```

See the new `STD_PRES_BINS` variable that hold the pressure bins definition.

The figure below shows the sub-sampling effect:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import cmocean

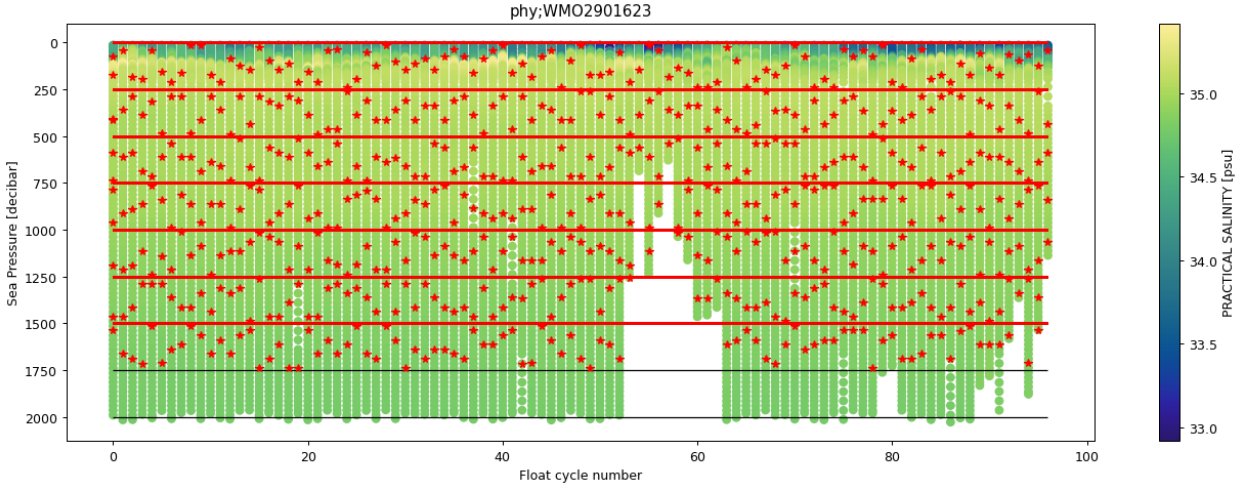
fig, ax = plt.subplots(figsize=(18,6))
ds.plot.scatter(x='CYCLE_NUMBER', y='PRES', hue='PSAL', ax=ax, cmap=cmocean.cm.haline)
plt.plot(ds_binned['CYCLE_NUMBER'], ds_binned['PRES'], 'r+')
plt.hlines(bins, ds['CYCLE_NUMBER'].min(), ds['CYCLE_NUMBER'].max(), color='k')
plt.hlines(ds_binned['STD_PRES_BINS'], ds_binned['CYCLE_NUMBER'].min(), ds_binned['CYCLE_
↪NUMBER'].max(), color='r')
plt.title(ds.attrs['Fetched_constraints'])
plt.gca().invert_yaxis()
```



The bin limits are shown with horizontal red lines, the original data are in the background colored scatter and the group-by pressure bins values are highlighted in red marks

The `select` option can take many different values, see the full documentation of *`Dataset.argo.groupby_pressure_bins()`* , for all the details. Let's show here results from the `random` sampling:

```
ds_binned = ds.argo.groupby_pressure_bins(bins=bins, select='random')
```



Filters

If you fetched data with the `expert` mode, you may want to use *filters* to help you curate the data.

- **QC flag filter**: *`Dataset.argo.filter_qc()`*. This method allows you to filter measurements according to QC flag values. This filter modifies all variables of the dataset.
- **Data mode filter**: *`Dataset.argo.filter_data_mode()`*. This method allows you to filter variables according to their data mode. This filter modifies the <PARAM> and <PARAM_QC> variables of the dataset.
- **OWC variables filter**: *`Dataset.argo.filter_scalib_pres()`*. This method allows you to filter variables according to OWC salinity calibration software requirements. This filter modifies pressure, temperature and salinity related variables of the dataset.

Complementary data

TEOS-10 variables

You can compute additional ocean variables from TEOS-10. The default list of variables is: 'SA', 'CT', 'SIG0', 'N2', 'PV', 'PTEMP' ('SOUND_SPEED', 'CNDC' are optional). Simply raise an issue to add a new one.

This can be done using the *Dataset.argo.teos10()* method and indicating the list of variables you want to compute:

```
In [16]: ds = DataFetcher().float(2901623).to_xarray()

In [17]: ds.argo.teos10(['SA', 'CT', 'PV'])
Out[17]:
<xarray.Dataset>
Dimensions:          (N_POINTS: 8341)
Coordinates:
  * N_POINTS         (N_POINTS) int64 0 1 2 3 4 5 ... 8336 8337 8338 8339 8340
    LATITUDE         (N_POINTS) float64 0.012 0.012 0.012 ... 3.388 3.388 3.388
    LONGITUDE        (N_POINTS) float64 92.28 92.28 92.28 ... 94.77 94.77 94.77
    TIME             (N_POINTS) datetime64[ns] 2010-05-14T03:35:00 ... 2013-0...
Data variables: (12/18)
    CYCLE_NUMBER     (N_POINTS) int64 0 0 0 0 0 0 0 0 ... 96 96 96 96 96 96 96
    DATA_MODE        (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D' 'D'
    DIRECTION        (N_POINTS) <U1 'D' 'D' 'D' 'D' 'D' ... 'A' 'A' 'A' 'A' 'A'
    PLATFORM_NUMBER  (N_POINTS) int64 2901623 2901623 ... 2901623 2901623
    POSITION_QC      (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    PRES             (N_POINTS) float64 17.0 25.0 35.0 ... 1.112e+03 1.137e+03
    ...               ...
    TEMP_ERROR       (N_POINTS) float64 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    TEMP_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    TIME_QC          (N_POINTS) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
    SA               (N_POINTS) float64 34.44 34.44 34.44 ... 35.09 35.09 35.08
    CT               (N_POINTS) float64 30.2 30.2 30.2 ... 6.068 6.078 5.959
    PV               (N_POINTS) float64 nan -1.78e-15 ... 1.573e-12 nan
Attributes:
    DATA_ID:              ARGO
    DOI:                  http://doi.org/10.17882/42182
    Fetched_from:         https://erddap.ifremer.fr/erddap
    Fetched_by:           docs
    Fetched_date:         2024/04/22
    Fetched_constraints:  WMO2901623
    Fetched_uri:          ['https://erddap.ifremer.fr/erddap/tabledap/ArgoFlo...
    history:              Variables filtered according to DATA_MODE; Variable...
```

```
In [18]: ds['SA']
Out[18]:
<xarray.DataArray 'SA' (N_POINTS: 8341)>
array([34.43600343, 34.43701333, 34.43703491, ..., 35.09205948,
       35.09221486, 35.08231586])
Coordinates:
  * N_POINTS   (N_POINTS) int64 0 1 2 3 4 5 6 ... 8335 8336 8337 8338 8339 8340
    LATITUDE   (N_POINTS) float64 0.012 0.012 0.012 0.012 ... 3.388 3.388 3.388
    LONGITUDE  (N_POINTS) float64 92.28 92.28 92.28 92.28 ... 94.77 94.77 94.77
```

(continues on next page)

(continued from previous page)

```
    TIME       (N_POINTS) datetime64[ns] 2010-05-14T03:35:00 ... 2013-01-01T0...
Attributes:
    long_name:      Absolute Salinity
    standard_name:  sea_water_absolute_salinity
    unit:           g/kg
```

Data models

By default **argopy** works with `xarray.Dataset` for Argo data fetcher, and with `pandas.DataFrame` for Argo index fetcher.

For your own analysis, you may prefer to switch from one to the other. This is all built in **argopy**, with the *argopy.DataFetcher.to_dataframe()* and *argopy.IndexFetcher.to_xarray()* methods.

```
In [19]: DataFetcher().profile(6902746, 34).to_dataframe()
Out[19]:
          CYCLE_NUMBER DATA_MODE  ... LONGITUDE                TIME
N_POINTS                          ...
0                   34         D  ...   -58.119 2017-12-20 06:58:00
1                   34         D  ...   -58.119 2017-12-20 06:58:00
2                   34         D  ...   -58.119 2017-12-20 06:58:00
3                   34         D  ...   -58.119 2017-12-20 06:58:00
4                   34         D  ...   -58.119 2017-12-20 06:58:00
...                ...       ...  ...       ...                 ...
104                 34         D  ...   -58.119 2017-12-20 06:58:00
105                 34         D  ...   -58.119 2017-12-20 06:58:00
106                 34         D  ...   -58.119 2017-12-20 06:58:00
107                 34         D  ...   -58.119 2017-12-20 06:58:00
108                 34         D  ...   -58.119 2017-12-20 06:58:00

[109 rows x 18 columns]
```

1.7.2 Saving data

Once you have your Argo data as `xarray.Dataset`, simply use the awesome possibilities of xarray like `xarray.Dataset.to_netcdf()` or `xarray.Dataset.to_zarr()`.

1.7.3 Data visualisation

From Data or Index fetchers

The `DataFetcher` and `IndexFetcher` come with a `plot` method to have a quick look to your data. This method can take *trajectory*, *profiler*, *dac* and *qc_altimetry* as arguments. All details are available in the `DataFetcher.plot` and `IndexFetcher.plot` class documentation.

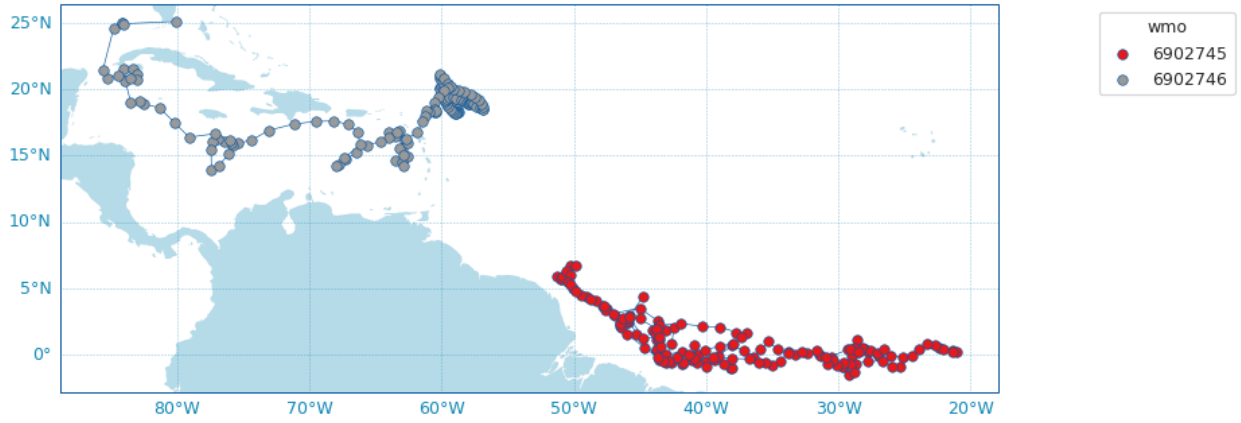
Below we demonstrate major plotting features.

Let's import the usual suspects:

```
from argopy import DataFetcher, IndexFetcher
```

Trajectories

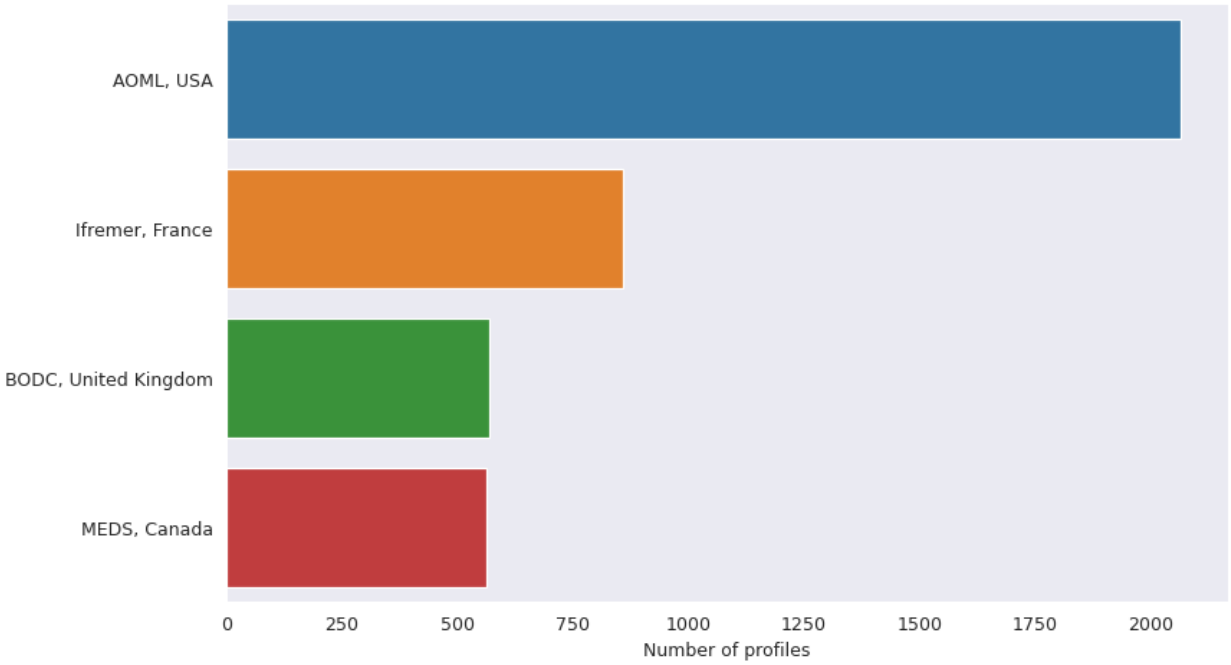
```
idx = IndexFetcher().float([6902745, 6902746]).load()
fig, ax = idx.plot('trajectory')
fig, ax = idx.plot()  # Trajectory is the default plot
```



Histograms on properties

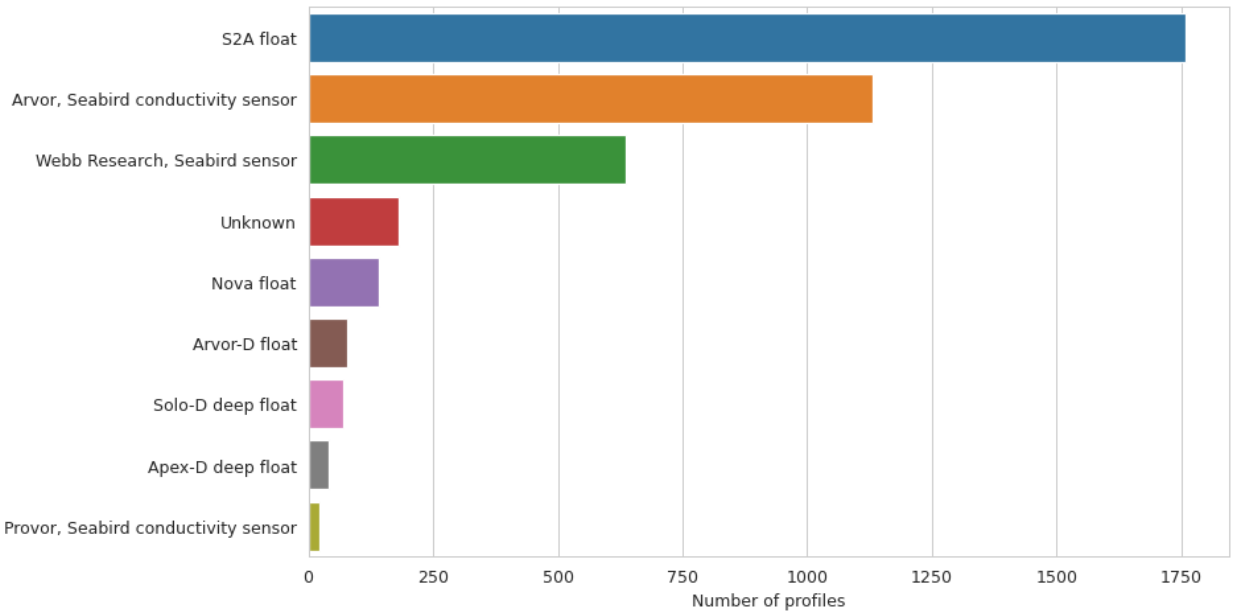
It is also possible to create horizontal bar plots for histograms on some data properties: *profiler* and *dac*:

```
idx = IndexFetcher().region([-80,-30,20,50,'2021-01','2021-08']).load()
fig, ax = idx.plot('dac')
```



If you have Seaborn installed, you can change the plot style:

```
fig, ax = idx.plot('profiler', style='whitegrid')
```



Dashboards

We provide shortcuts to third-party online dashboards that can help you visualise float or profile data. When working in Jupyter notebooks, you can insert a dashboard in a cell, or if you don't, you can get the url toward the dashboard to open it elsewhere.

We provide access to the Euro-Argo ERIC, Ocean-OPS, Argovis and BGC dashboards with the option `type`. See *dashboard()* for all the options.

Summary of all available dashboards:

Type name	base	float	profile
"data", "ea"	X	X	X
"meta"	X	X	X
"bgc"	X	X	X
"ocean-ops", "op"	X	X	
"coriolis", "cor"		X	
"argovis"	X	X	X

Examples:

Default dashboard

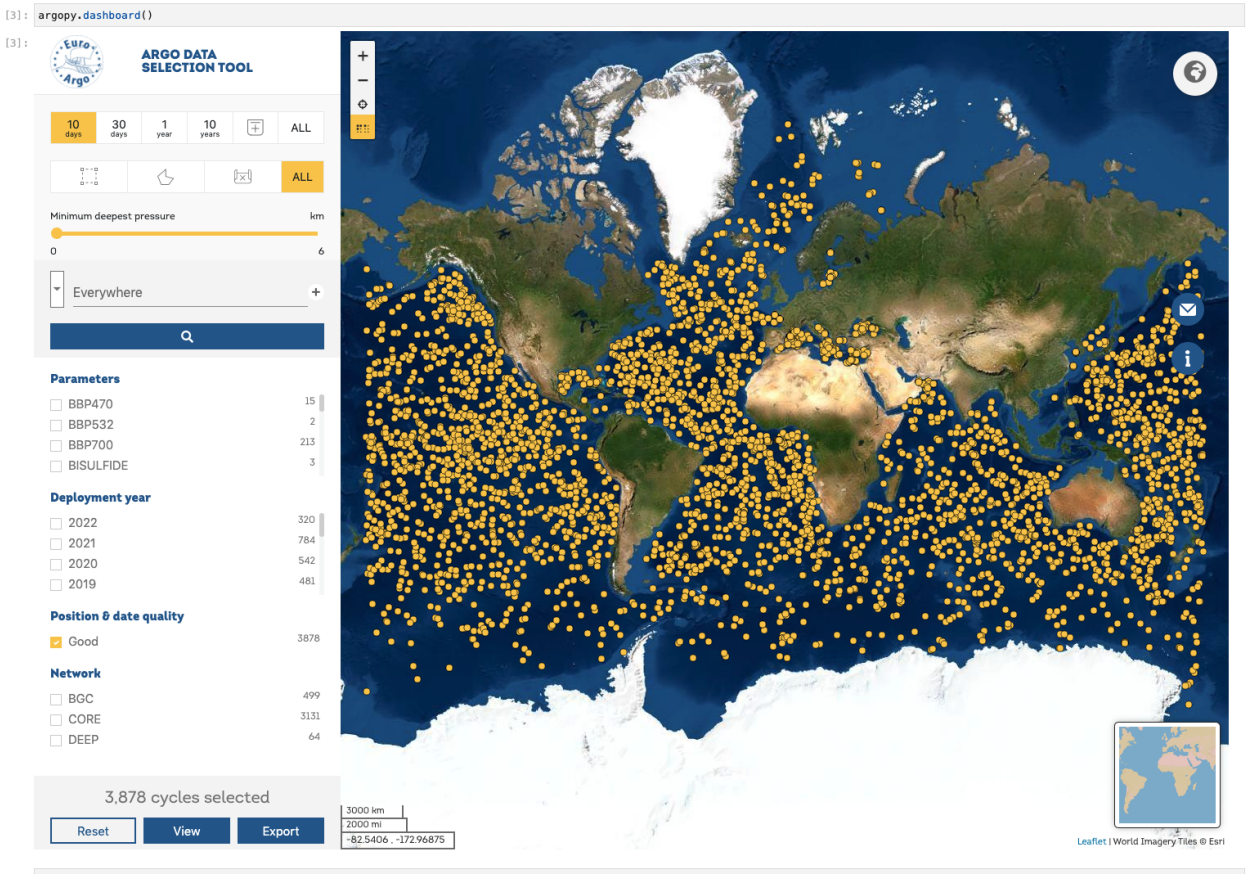
For floats

For profiles

For **bgc** profiles

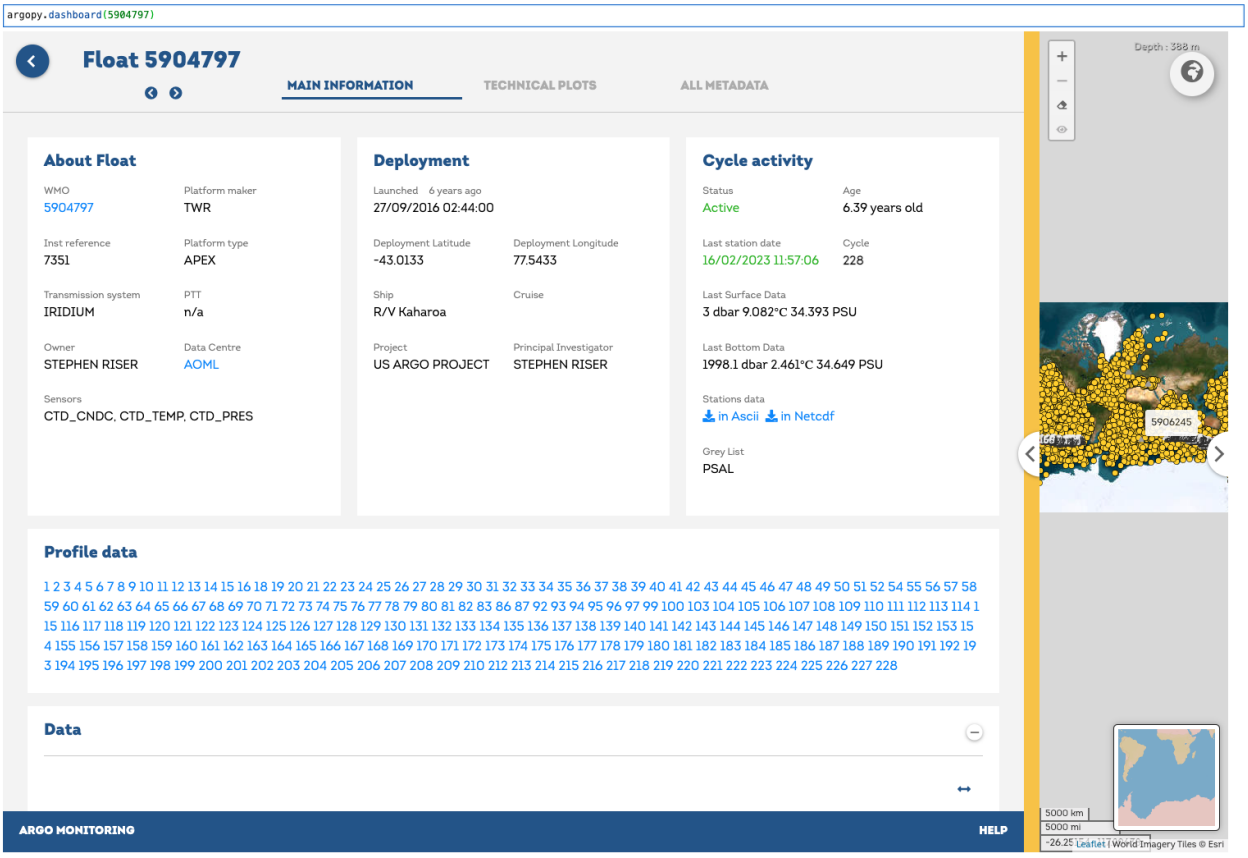
Open the default dashboard like this:

```
argopy.dashboard()
```



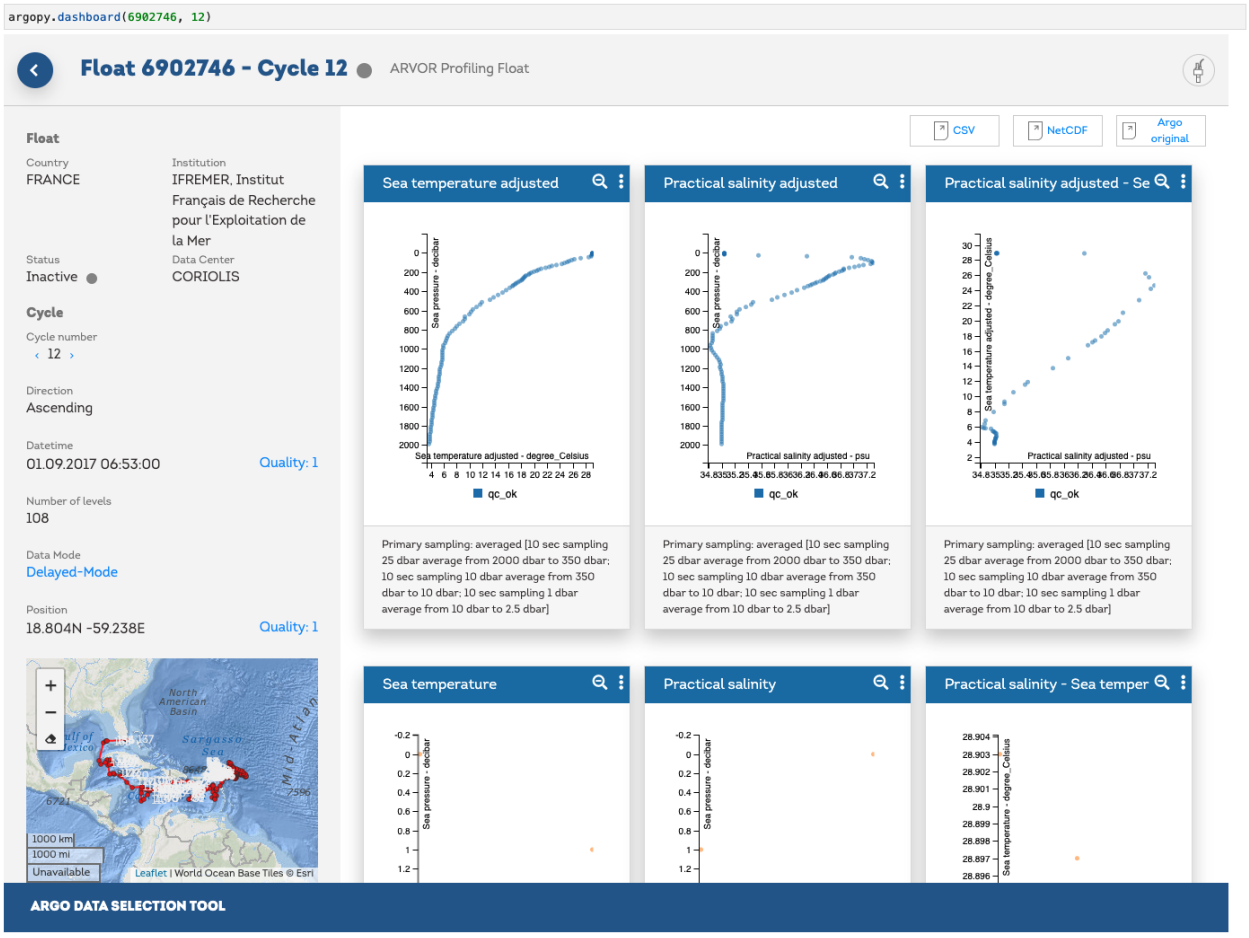
For a specific float, just provide its WMO:

```
argopy.dashboard(5904797)
```



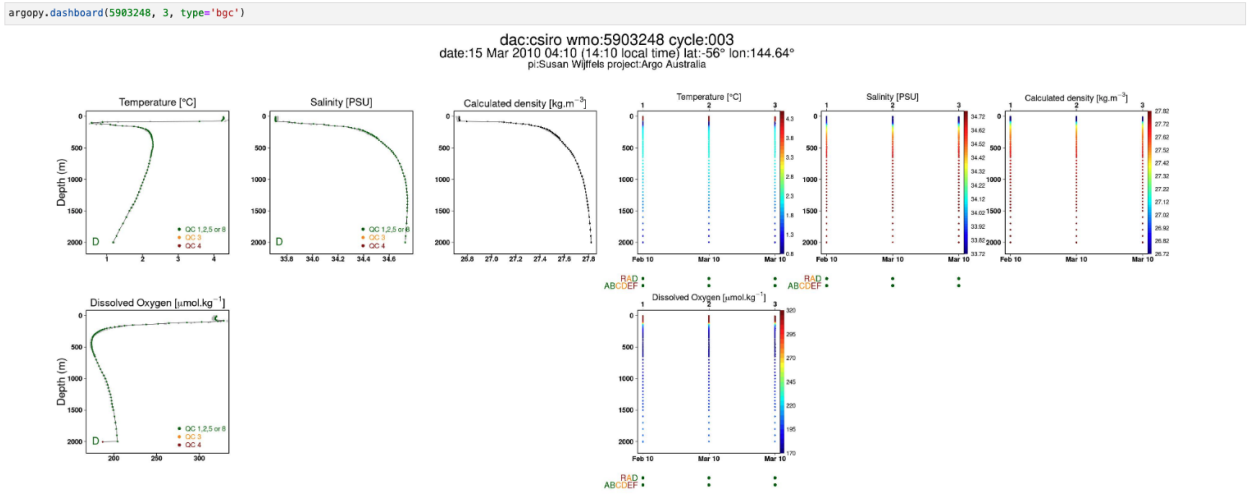
For a specific float profile, provide its WMO and cycle number:

```
argopy.dashboard(6902746, 12)
```



and for a BGC float, change the `type` option to `bgc`:

```
argopy.dashboard(5903248, 3, type='bgc')
```



Note: Dashboards can be open at the package level or from data fetchers. So that we have the following equivalence:

```
argopy.dashboard(WMO)
# similar to:
DataFetcher().float(WMO).dashboard()
```

and:

```
argopy.dashboard(WMO, CYC)
# similar to:
DataFetcher().profile(WMO, CYC).dashboard()
```

Scatter Maps

The *argopy.plot.scatter_map* utility function is dedicated to making maps with Argo profile positions coloured according to specific variables: **a scatter map**.

Profiles colouring is finely tuned for some variables: QC flags, Data Mode and Deployment Status. By default, floats trajectories are always shown, but this can be changed with the `traj` boolean option.

Note that the *argopy.plot.scatter_map* integrates seamlessly with **argopy** *Index of profiles* `pandas.DataFrame` and `xarray.Dataset` *collection of profiles*. However, all default arguments can be overwritten so that it should work with other data models.

Let's import the usual suspects and some data to work with.

```
from argopy.plot import scatter_map
from argopy import DataFetcher, OceanOPSDeployments

ArgoSet = DataFetcher(mode='expert').float([6902771, 4903348]).load()
ds = ArgoSet.data.argo.point2profile()
df = ArgoSet.index

df_deployment = OceanOPSDeployments([-90, 0, 0, 90]).to_dataframe()
```

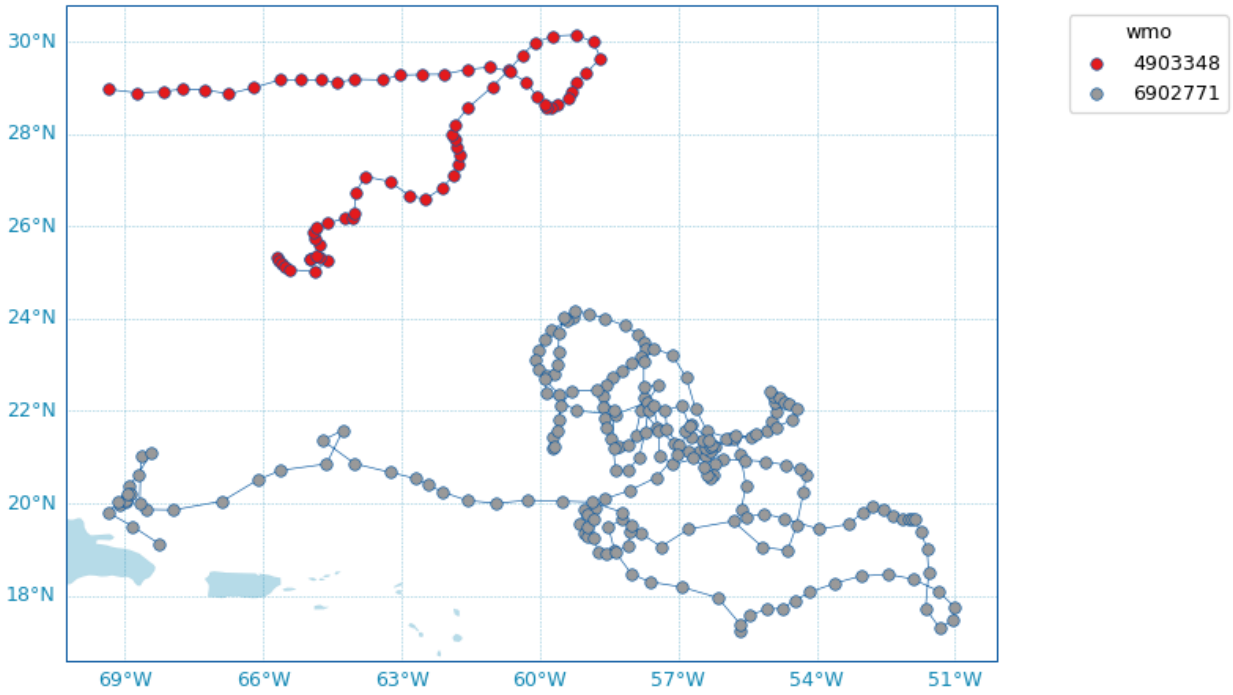
And see in the examples below how it can be used and tuned.

Default scatter map for trajectories

By default, the *argopy.plot.scatter_map()* function will try to plot a trajectory map, i.e. a map where profile points are of the same color for each floats and joined by a simple line.

Note: If Cartopy is installed, the `argopy.plot.plot_trajectory()` called by *DataFetcher.plot* and *IndexFetcher.plot* with the `trajectory` option will rely on the scatter map described here.

```
scatter_map(df)
```

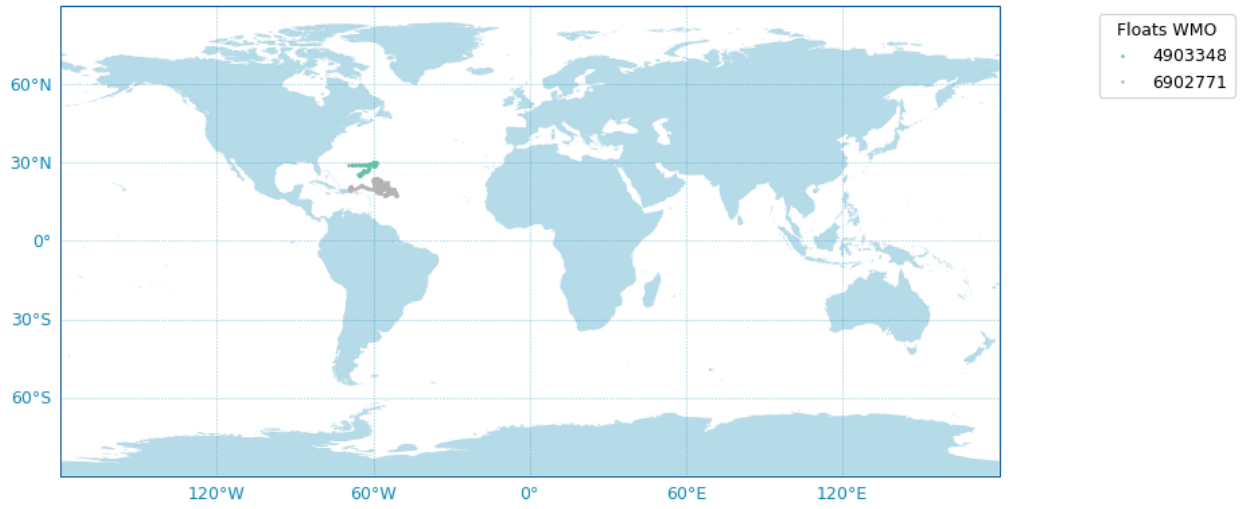


Arguments can be passed explicitly as well:

```
scatter_map(df,
            x='longitude',
            y='latitude',
            hue='wmo',
            cmap='Set1',
            traj_axis='wmo')
```

Some options are available to customise the plot, for instance:

```
fig, ax = scatter_map(df,
                      figsize=(10,6),
                      set_global=True,
                      markersize=2,
                      markeredgecolor=None,
                      legend_title='Floats WMO',
                      cmap='Set2')
```



Use predefined Argo Colors

The *argopy.plot.scatter_map* function uses the *ArgoColors* utility class to better resolve discrete colormaps of known variables. The colormap is automatically guessed using the hue argument. Here are some examples.

Parameter Data Mode

QC flag

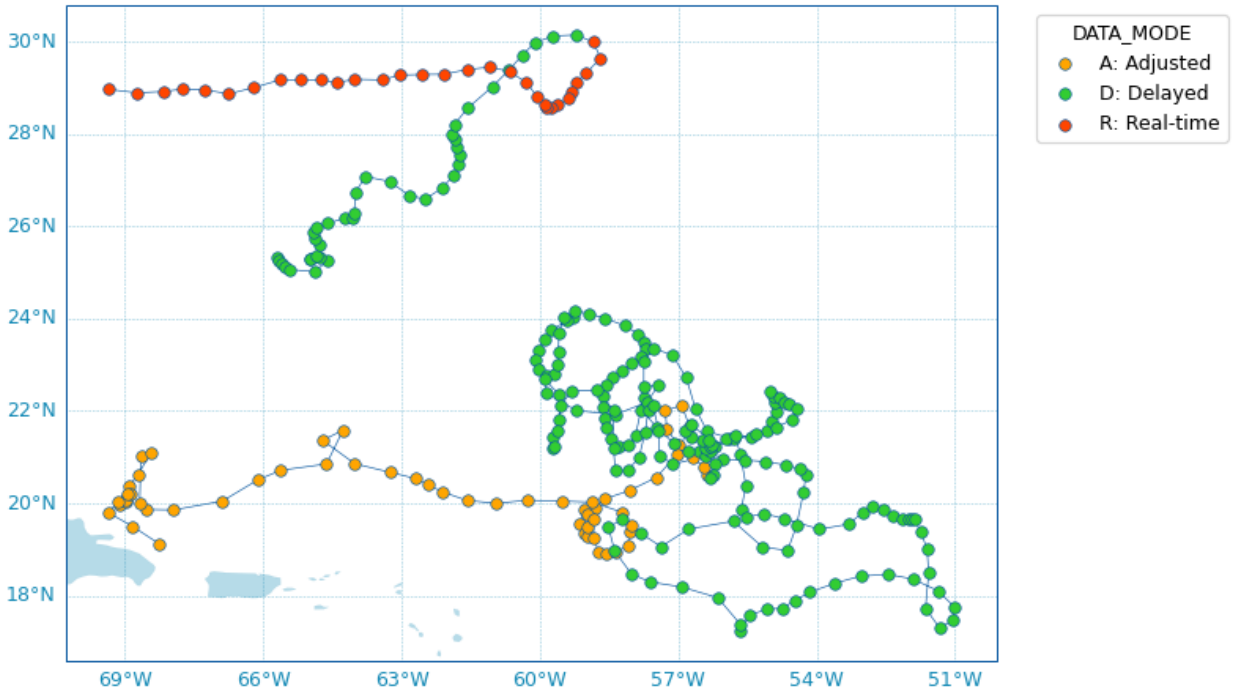
Deployment status

Using guess mode for arguments:

```
scatter_map(ds, hue='DATA_MODE')
```

or more explicitly:

```
scatter_map(ds,
            x='LONGITUDE',
            y='LATITUDE',
            hue='DATA_MODE',
            cmap='data_mode',
            traj_axis='PLATFORM_NUMBER')
```

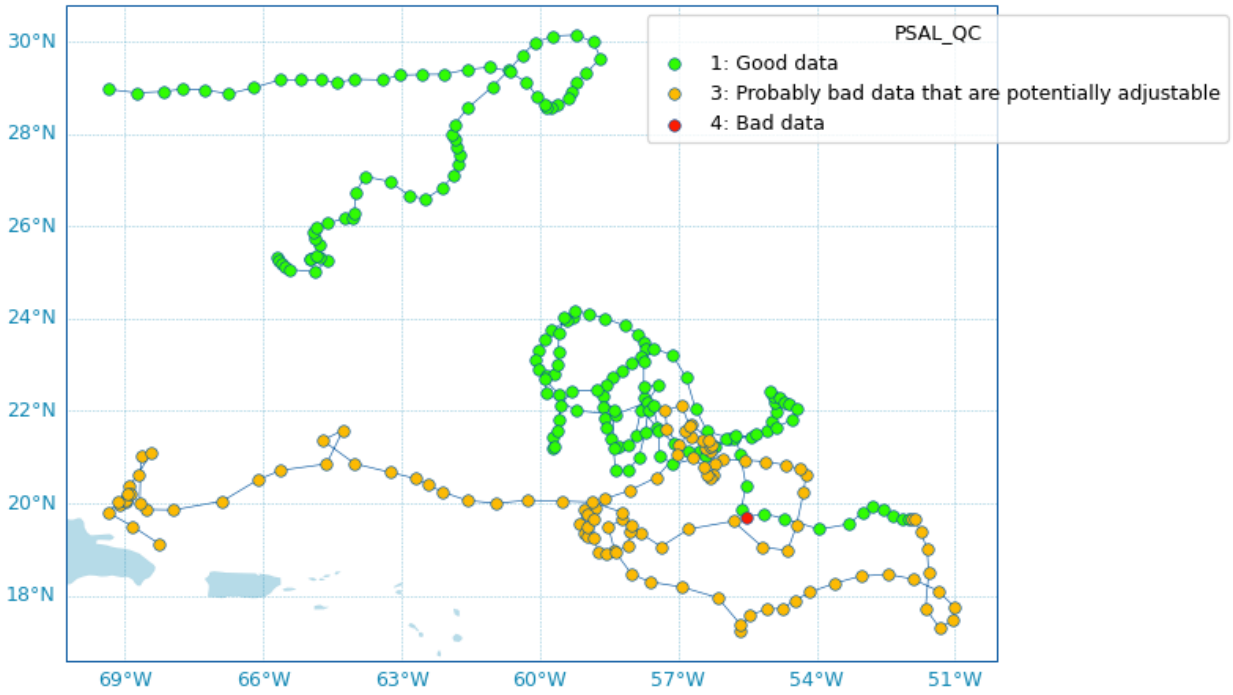


Since QC flags are given for each measurements, we need to select a specific depth levels for this plot:

```
scatter_map(ds, hue='PSAL_QC')
```

using guess mode for arguments, or more explicitly:

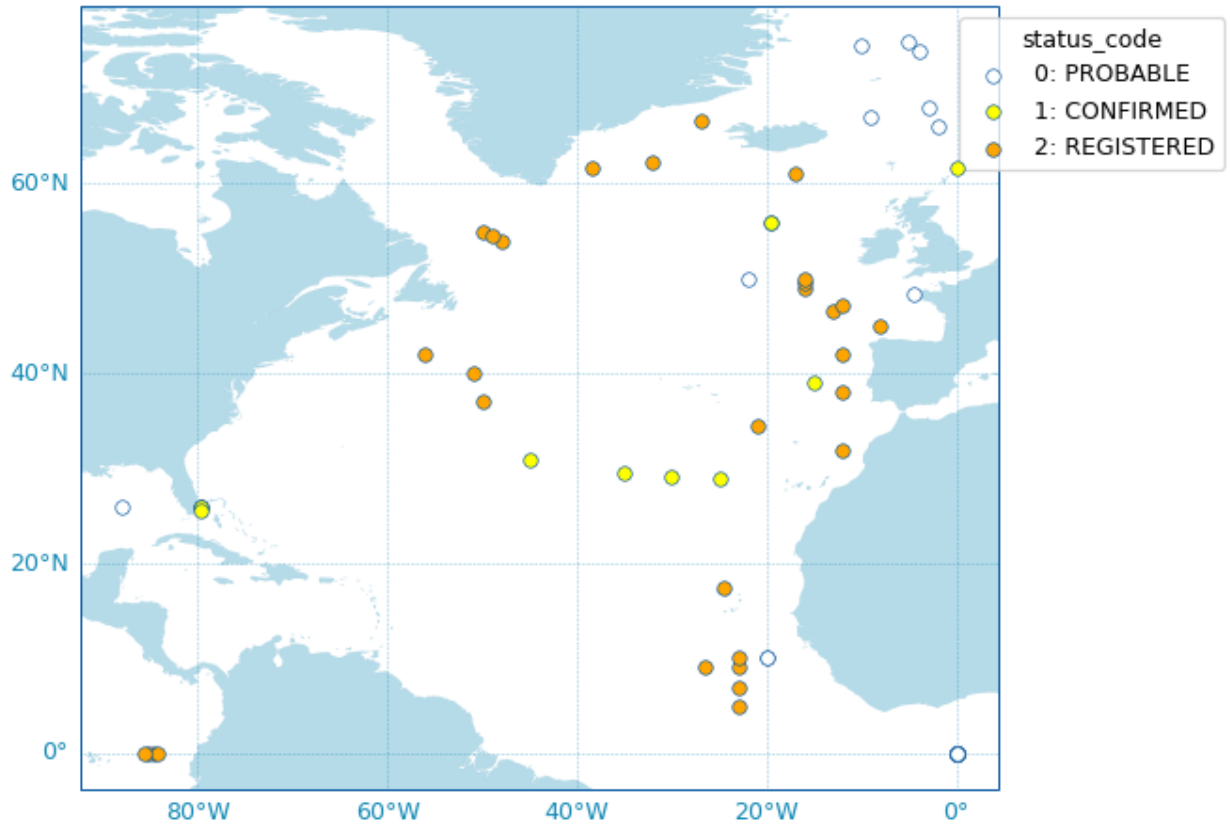
```
scatter_map(ds.isel(N_LEVELS=0),
            x='LONGITUDE',
            y='LATITUDE',
            hue='PSAL_QC',
            cmap='qc',
            traj_axis='PLATFORM_NUMBER')
```



Hint: Since the `PSAL_QC` variable has more than vertical level, we need to select which one to use for the plot.

For the deployment status, there is only one point for each float, so we can make a faster plot by not using the `traj` option.

```
scatter_map(df_deployment, hue='status_code', traj=False)
```

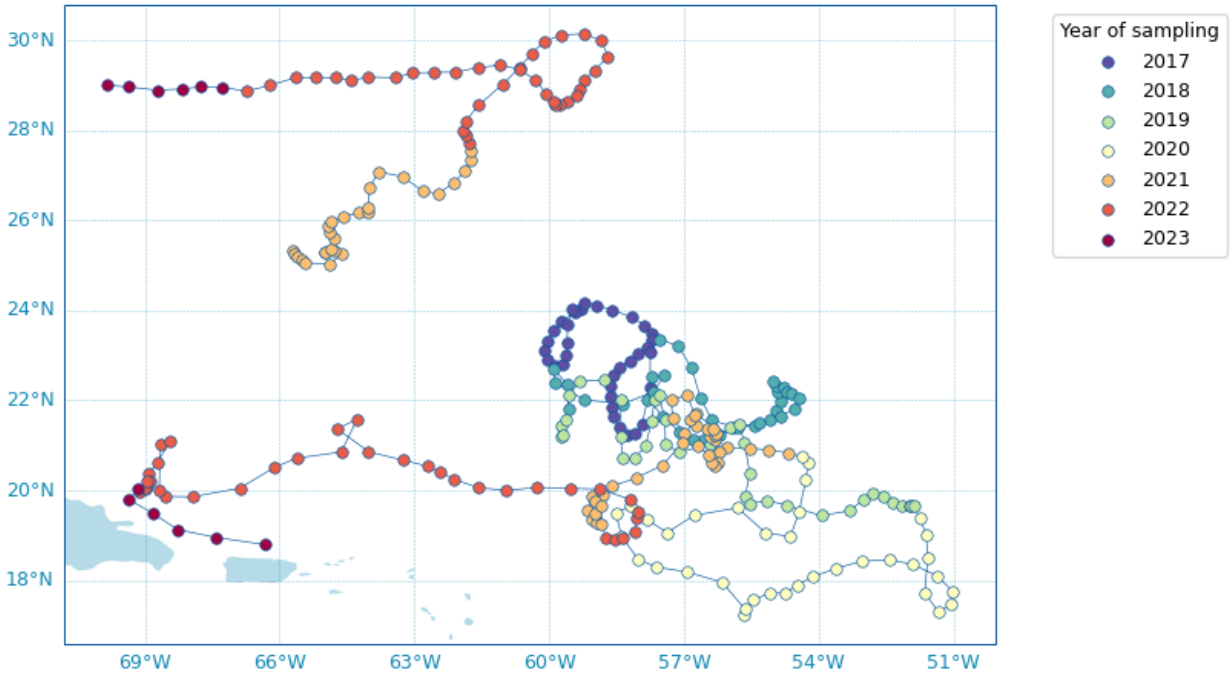


Use any colormap

Beyond the predefined set of Argo colors, one can use any colormap that can be discretesized.

In the example below, we plot profile years of sampling using the reverse `Spectral` colormap:

```
ds['year'] = ds['TIME.year']  # Add new variable to the dataset
scatter_map(ds,
            hue='year',
            cmap='Spectral_r',
            legend_title='Year of sampling')
```



Argo colors

For your own plot methods, **argopy** provides the *ArgoColors* utility class to better resolve discrete colormaps of known Argo variables.

The class *ArgoColors* is used to get a discrete colormap (available with the `cmap` attribute), as a `matplotlib.colors.LinearSegmentedColormap`.

The *Use predefined Argo Colors* section above gives examples of the available colormaps that are also summarized here:

Parameter Data Mode

Quality control flag scale

Deployment status

Months

```
ArgoColors('data_mode')
```

Argo Data-Mode

Names: data_mode, datamode, dm

R	Real-time
A	Adjusted
D	Delayed
	FillValue

```
In [1]: ArgoColors('data_mode').definition
Out[1]:
{'name': 'Argo Data-Mode',
 'aka': ['datamode', 'dm'],
 'constructor': <bound method ArgoColors._colormap_datamode of <argopy.plot.argo_colors.
↪ArgoColors object at 0x7fd6c7461a00>>,
 'ticks': ['R', 'A', 'D', ' '],
 'ticklabels': ['Real-time', 'Adjusted', 'Delayed', 'FillValue']}
```

```
ArgoColors('qc_flag')
```

Quality control flag scale

Names: qc, qc_flag, quality_control, quality_control_flag, quality_control_flag_scale

0	No QC performed
1	Good data
2	Probably good data
3	Probably bad data that are potentially adjustable
4	Bad data
5	Value changed
6	Not used
7	Not used
8	Estimated value
9	Missing value

```
In [2]: ArgoColors('qc_flag').definition
Out[2]:
{'name': 'Quality control flag scale',
 'aka': ['qc_flag',
  'quality_control',
  'quality_control_flag',
  'quality_control_flag_scale'],
 'constructor': <bound method ArgoColors._colormap_quality_control_flag of <argopy.plot.
↪argo_colors.ArgoColors object at 0x7fd6b3b7dbe0>>,
 'ticks': array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]),
 'ticklabels': ['No QC performed',
  'Good data',
  'Probably good data',
  'Probably bad data that are potentially adjustable',
  'Bad data',
  'Value changed',
  'Not used',
  'Not used',
  'Estimated value',
  'Missing value']}
```

```
ArgoColors('deployment_status')
```

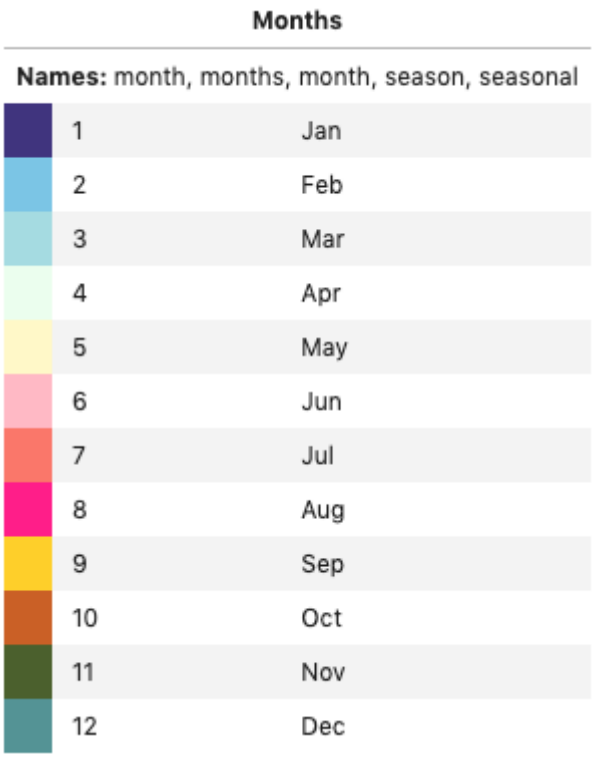
Deployment status

Names: deployment_status, deployment_code, deployment_id, ptfstatus.id, ptfstatus, status_code

0	PROBABLE
1	CONFIRMED
2	REGISTERED
6	OPERATIONAL
4	INACTIVE
5	CLOSED

```
In [3]: ArgoColors('deployment_status').definition
Out[3]:
{'name': 'Deployment status',
 'aka': ['deployment_code',
  'deployment_id',
  'ptfstatus.id',
  'ptfstatus',
  'status_code'],
 'constructor': <bound method ArgoColors._colormap_deployment_status of <argopy.plot.
↪argo_colors.ArgoColors object at 0x7fd6a6fd2ca0>>,
 'ticks': [0, 1, 2, 6, 4, 5],
 'ticklabels': ['PROBABLE',
  'CONFIRMED',
  'REGISTERED',
  'OPERATIONAL',
  'INACTIVE',
  'CLOSED']}
```

```
ArgoColors('months')
```



Months

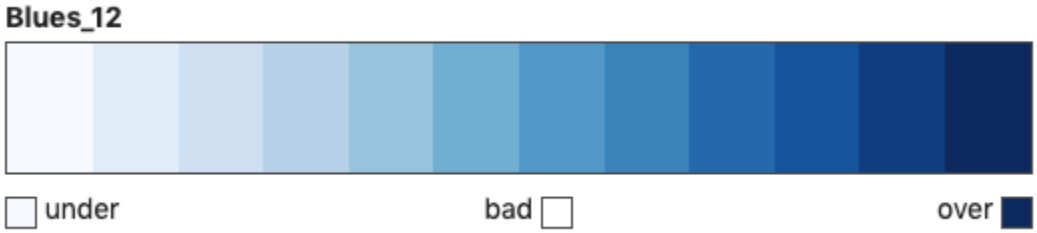
Names: month, months, month, season, seasonal

1	Jan
2	Feb
3	Mar
4	Apr
5	May
6	Jun
7	Jul
8	Aug
9	Sep
10	Oct
11	Nov
12	Dec

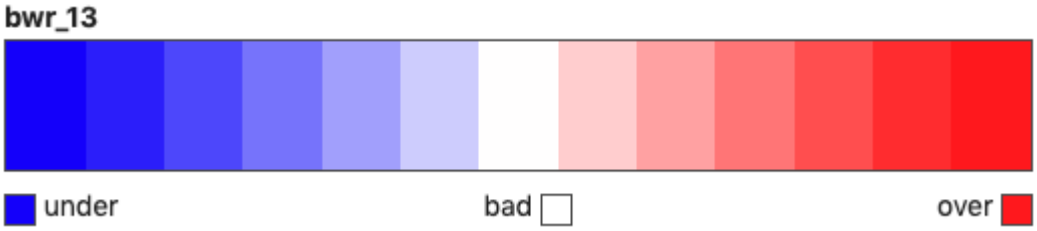
```
In [4]: ArgoColors('months').definition
Out[4]:
{'name': 'Months',
 'aka': ['months', 'month', 'season', 'seasonal'],
 'constructor': <bound method ArgoColors._colormap_month of <argopy.plot.argo_colors.
↪ArgoColors object at 0x7fd6c7461af0>>,
 'ticks': array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12]),
 'ticklabels': ['Jan',
  'Feb',
  'Mar',
  'Apr',
  'May',
  'Jun',
  'Jul',
  'Aug',
  'Sep',
  'Oct',
  'Nov',
  'Dec']}
```

Note that *ArgoColors* can also be used to discretise any colormap:

```
ArgoColors('Blues')
```




```
ArgoColors('bwr', N=13)
```



1.7.4 Data quality control

Hint: **argopy** comes with methods to help you quality control measurements. This section is probably intended for **expert** users.

Most of these methods are available through the `xarray.Dataset` accessor namespace `argo`. This means that if your dataset is *ds*, then you can use *ds.argo* to access more **argopy** functionalities.

Let's start with standard import:

```
In [1]: import argopy

In [2]: argopy.clear_cache()

In [3]: argopy.reset_options()

In [4]: from argopy import DataFetcher
```

Salinity calibration

The Argo salinity calibration method is called [OWC], after the names of the core developers: Breck Owens, Anny Wong and Cecile Cabanes. Historically, the OWC method has been implemented in Matlab . More recently a python version has been developed.

Preprocessing data

At this point, both OWC software take as input a pre-processed version of the Argo float data to evaluate/calibrate.

argopy is able to perform this preprocessing and to create a *float source* data to be used by OWC software. This is made by *Dataset.argo.create_float_source()*.

First, you would need to fetch the Argo float data you want to calibrate, in `expert` mode:

```
In [5]: ds = DataFetcher(mode='expert').float(6902766).load().data
```

Then, to create the float source data, you call the method and provide a folder name to save output files:

```
In [6]: ds.argo.create_float_source("float_source")
```

This will create the `float_source/6902766.mat` Matlab files to be set directly in the configuration file of the OWC software. This routine implements the same pre-processing as in the Matlab version (which is hosted on this repo and ran with this routine). All the detailed steps of this pre-processing are given in the *Dataset.argo.create_float_source()* API page.

Note: If the dataset contains data from more than one float, several Matlab files are created, one for each float. This will allow you to prepare data from a collection of floats.

If you don't specify a path name, the method returns a dictionary with the float WMO as keys and pre-processed data as `xarray.Dataset` as values.

```
In [7]: ds_source = ds.argo.create_float_source()

In [8]: ds_source
Out[8]:
{6902766: <xarray.Dataset>
 Dimensions:     (m: 203, n: 247)
 Coordinates:
   * m           (m) int64 0 1 2 3 4 5 6 7 8 ... 195 196 197 198 199 200 201 202
   * n           (n) int64 0 1 2 3 4 5 6 7 8 ... 239 240 241 242 243 244 245 246
 Data variables:
     PRES        (m, n) float32 9.4 8.9 9.4 9.8 9.9 ... nan nan nan 2.028e+03 nan
     TEMP        (m, n) float32 24.57 24.91 24.72 25.1 ... nan nan 3.695 nan
     PTMP        (m, n) float64 24.57 24.91 24.72 25.1 ... nan nan 3.529 nan
     SAL         (m, n) float64 37.37 37.35 37.45 37.33 ... nan nan 35.01 nan
     PROFILE_NO  (n) int64 1 2 3 4 5 6 7 8 9 ... 240 241 242 243 244 245 246 247
     DATES       (n) float64 2.017e+03 2.017e+03 ... 2.024e+03 2.024e+03
     LAT         (n) float64 20.34 20.43 20.55 20.7 ... 22.51 22.69 22.95 23.2
     LONG        (n) float64 310.4 309.9 309.5 309.2 ... 305.0 305.0 304.9 304.8}
```

See all options available for this method here: *Dataset.argo.create_float_source()*.

The method partially relies on two others:

- *Dataset.argo.filter_scalib_pres()*: to filter variables according to OWC salinity calibration software requirements. This filter modifies pressure, temperature and salinity related variables of the dataset.
- *Dataset.argo.groupby_pressure_bins()*: to sub-sampled measurements by pressure bins. This is an excellent alternative to the *Dataset.argo.interp_std_levels()* to avoid interpolation and preserve values of raw measurements while at the same time aligning measurements along approximately similar pressure levels (depending on the size of the bins). See more description at here: *Pressure levels: Group-by bins*.

Running the calibration

Please refer to the OWC python software documentation.

Listing 1: Typical OWC workflow example

```
import os, shutil
from pathlib import Path

import pyowc as owc
from argopy import DataFetcher

# Define float to calibrate:
FLOAT_NAME = "6903010"

# Set-up where to save OWC analysis results:
results_folder = './analysis/%s' % FLOAT_NAME
Path(results_folder).mkdir(parents=True, exist_ok=True)
shutil.rmtree(results_folder)  # Clean up folder content
Path(os.path.sep.join([results_folder, 'float_source'])).mkdir(parents=True, exist_
↪ok=True)
Path(os.path.sep.join([results_folder, 'float_calib'])).mkdir(parents=True, exist_
↪ok=True)
Path(os.path.sep.join([results_folder, 'float_mapped'])).mkdir(parents=True, exist_
↪ok=True)
Path(os.path.sep.join([results_folder, 'float_plots'])).mkdir(parents=True, exist_
↪ok=True)

# fetch the default configuration and parameters
USER_CONFIG = owc.configuration.load()

# Fix paths to run at Ifremer:
for k in USER_CONFIG:
    if "FLOAT" in k and "data/" in USER_CONFIG[k][0:5]:
        USER_CONFIG[k] = os.path.abspath(USER_CONFIG[k].replace("data", results_folder))
USER_CONFIG['CONFIG_DIRECTORY'] = os.path.abspath('../data/constants')
USER_CONFIG['HISTORICAL_DIRECTORY'] = os.path.abspath(
    '/Volumes/OWC/CLIMATOLOGY/')  # where to find ARGO_for_DMQC_2020V03 and CTD_for_DMQC_
↪2021V01 folders
USER_CONFIG['HISTORICAL_ARGO_PREFIX'] = 'ARGO_for_DMQC_2020V03/argo_'
USER_CONFIG['HISTORICAL_CTD_PREFIX'] = 'CTD_for_DMQC_2021V01/ctd_'
print(owc.configuration.print_cfg(USER_CONFIG))

# Create float source data with argopy:
fetcher_for_real = DataFetcher(src='localftp', cache=True, mode='expert').float(FLOAT_
```

(continues on next page)

(continued from previous page)

```
↪NAME)
fetcher_sample = DataFetcher(src='localftp', cache=True, mode='expert').profile(FLOAT_
↪NAME, [1,
                                                                                    ⌋
↪    2])  # To reduce execution time for demo
ds = fetcher_sample.load().data
ds.argo.create_float_source(path=USER_CONFIG['FLOAT_SOURCE_DIRECTORY'], force='default')

# Prepare data for calibration: map salinity on theta levels
owc.calibration.update_salinity_mapping("", USER_CONFIG, FLOAT_NAME)

# Set the calseries parameters for analysis and line fitting
owc.configuration.set_calseries("", FLOAT_NAME, USER_CONFIG)

# Calculate the fit of each break and calibrate salinities
owc.calibration.calc_piecewisefit("", FLOAT_NAME, USER_CONFIG)

# Results figures
owc.plot.dashboard("", FLOAT_NAME, USER_CONFIG)
```

OWC references

"An improved calibration method for the drift of the conductivity sensor on autonomous CTD profiling floats by –S climatology". Deep-Sea Research Part I: Oceanographic Research Papers, 56(3), 450-457, 2009. https://doi.org/10.1016/j.dsr.2008.09.008

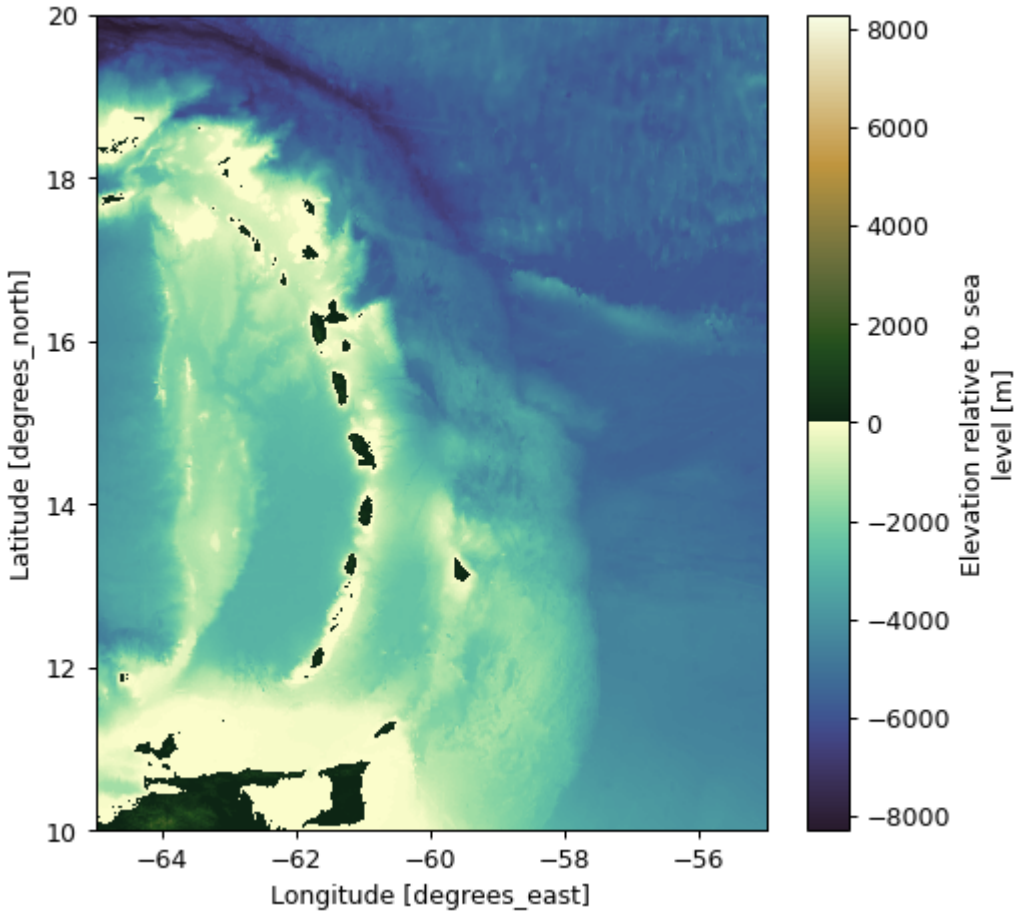
"Improvement of bias detection in Argo float conductivity sensors and its application in the North Atlantic". Deep-Sea Research Part I: Oceanographic Research Papers, 114, 128-136, 2016. https://doi.org/10.1016/j.dsr.2016.05.007

Trajectories

Topography

For some QC of trajectories, it can be useful to easily get access to the topography. This can be done with the **argopy** utility *TopoFetcher*:

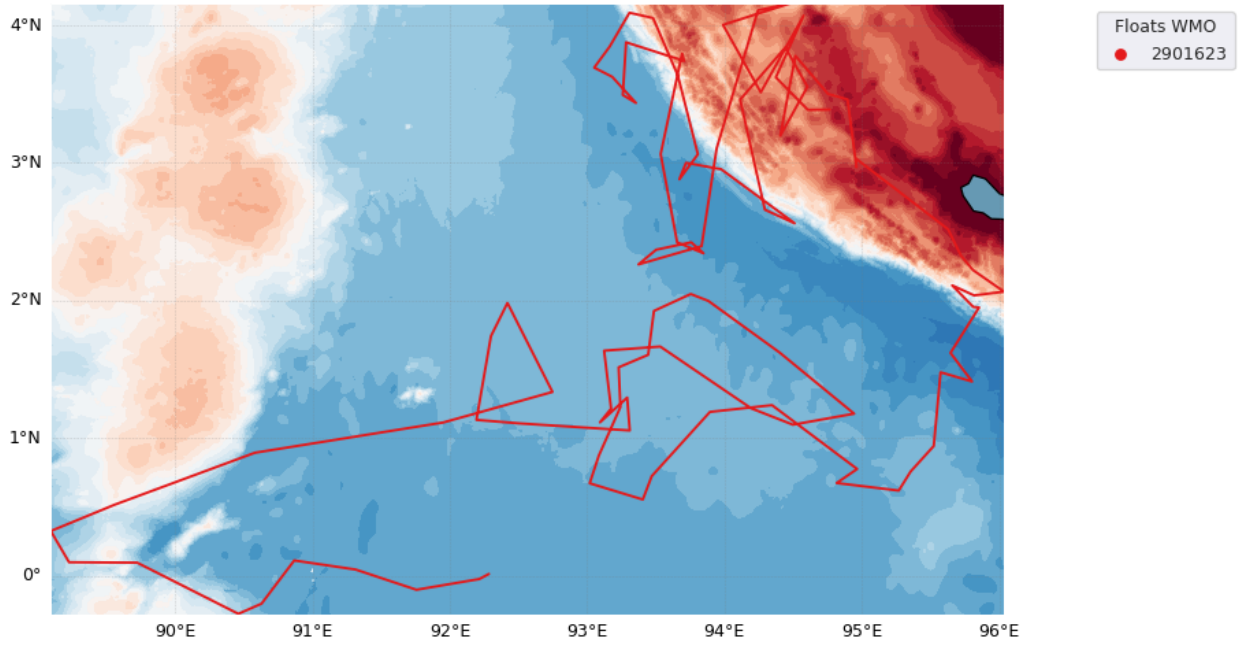
```
from argopy import TopoFetcher
box = [-65, -55, 10, 20]
ds = TopoFetcher(box, cache=True).to_xarray()
```



Combined with the fetcher property `domain`, it now becomes easy to superimpose float trajectory with topography:

```
fetcher = ArgoDataFetcher().float(2901623)
ds = TopoFetcher(fetcher.domain[0:4], cache=True).to_xarray()
```

```
fig, ax = loader.plot('trajectory', figsize=(10, 10))
ds['elevation'].plot.contourf(levels=np.arange(-6000,0,100), ax=ax, add_colorbar=False)
```



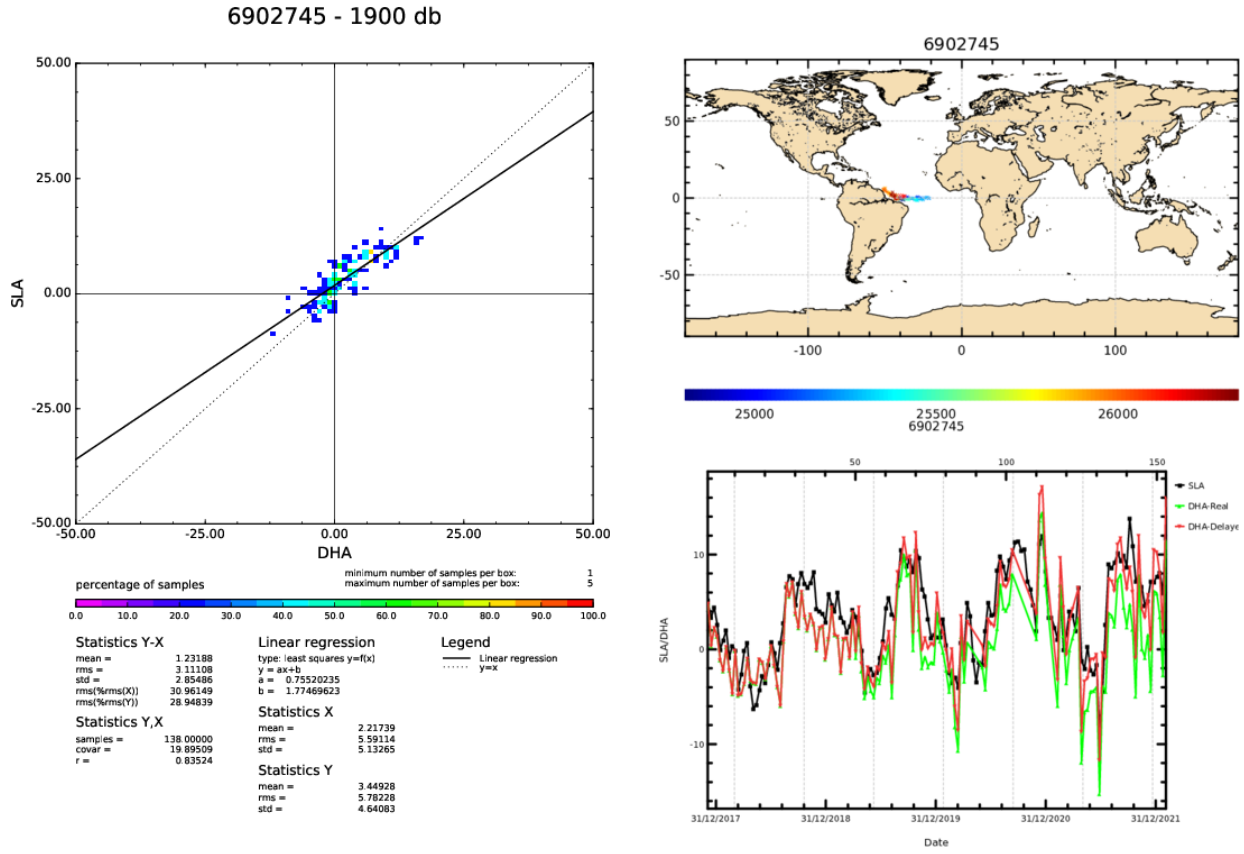
Note: The *TopoFetcher* can return a lower resolution topography with the `stride` option. See the *argopy.TopoFetcher* full documentation for all the details.

Altimetry

Satellite altimeter measurements can be used to check the quality of the Argo profiling floats time series. The method compares collocated sea level anomalies from altimeter measurements and dynamic height anomalies calculated from Argo temperature and salinity profiles for each Argo float time series [Guinehut2008]. This method is performed routinely by CLS and results are made available online.

argopy provides a simple access to this QC analysis with an option to the data and index fetchers *DataFetcher.plot()* methods that will insert the CLS Satellite Altimeter report figure on a notebook cell.

```
fetcher = ArgoDataFetcher().float(6902745)
fetcher.plot('qc_altimetry', embed='list')
```



See all details about this method here: `argopy.plot.open_sat_altim_report()`

References

1.8 Argo meta-data

1.8.1 Index of profiles

Since the Argo measurements dataset is quite complex, it comes with a collection of index files, or lookup tables with meta data. These index help you determine what you can expect before retrieving the full set of measurements.

argopy provides two methods to work with Argo index files: one is high-level and works like the data fetcher, the other is low-level and works like a "store".

Fetcher: High-level Argo index access

argopy has a specific fetcher for index files:

```
In [1]: from argopy import IndexFetcher as ArgoIndexFetcher
```

You can use the Index fetcher with the `region` or `float` access points, similarly to data fetching:

```
In [2]: idx = ArgoIndexFetcher(src='gdac').float(2901623).load()

In [3]: idx.index
Out[3]:
                                      file  ...                                  ␣
↪profiler
0    nmdis/2901623/profiles/R2901623_000.nc  ...  PROVOR float with SBE conductivity␣
↪sensor
1   nmdis/2901623/profiles/R2901623_000D.nc  ...  PROVOR float with SBE conductivity␣
↪sensor
2    nmdis/2901623/profiles/R2901623_001.nc  ...  PROVOR float with SBE conductivity␣
↪sensor
3    nmdis/2901623/profiles/R2901623_002.nc  ...  PROVOR float with SBE conductivity␣
↪sensor
4    nmdis/2901623/profiles/R2901623_003.nc  ...  PROVOR float with SBE conductivity␣
↪sensor
..                                      ...  ...                                   .
↪..
93   nmdis/2901623/profiles/R2901623_092.nc  ...  PROVOR float with SBE conductivity␣
↪sensor
94   nmdis/2901623/profiles/R2901623_093.nc  ...  PROVOR float with SBE conductivity␣
↪sensor
95   nmdis/2901623/profiles/R2901623_094.nc  ...  PROVOR float with SBE conductivity␣
↪sensor
96   nmdis/2901623/profiles/R2901623_095.nc  ...  PROVOR float with SBE conductivity␣
↪sensor
97   nmdis/2901623/profiles/R2901623_096.nc  ...  PROVOR float with SBE conductivity␣
↪sensor

[98 rows x 12 columns]
```

Alternatively, you can use *argopy.IndexFetcher.to_dataframe()*:

```
In [4]: idx = ArgoIndexFetcher(src='gdac').float(2901623)

In [5]: df = idx.to_dataframe()
```

The difference is that with the *load* method, data are stored in memory and not fetched on every call to the *index* attribute.

The index fetcher has pretty much the same methods than the data fetchers. You can check them all here: `argopy.fetchers.ArgoIndexFetcher`.

Store: Low-level Argo Index access

The IndexFetcher shown above is a user-friendly layer on top of our internal Argo index file store. But if you are familiar with Argo index files and/or cares about performances, you may be interested in using directly the Argo index store *ArgoIndex*.

If Pyarrow is installed, this store will rely on `pyarrow.Table` as internal storage format for the index, otherwise it will fall back on `pandas.DataFrame`. Loading the full Argo profile index takes about 2/3 secs with Pyarrow, while it can take up to 6/7 secs with Pandas.

All index store methods and properties are documented in *ArgoIndex*.

Index file supported

The table below summarize the **argopy** support status of all Argo index files:

Table 3: **argopy** GDAC index file support status

	Index file	Supported
Profile	ar_index_global_prof.txt	
Synthetic-Profile	argo_synthetic-profile_index.txt	
Bio-Profile	argo_bio-profile_index.txt	
Trajectory	ar_index_global_traj.txt	
Bio-Trajectory	argo_bio-traj_index.txt	
Metadata	ar_index_global_meta.txt	
Technical	ar_index_global_tech.txt	
Greylist	ar_greylist.txt	

Index files support can be added on demand. Click here to raise an issue if you'd like to access other index files.

Usage

You create an index store with default or custom options:

```
In [6]: from argopy import ArgoIndex

In [7]: idx = ArgoIndex()

# or:
# ArgoIndex(index_file="argo_bio-profile_index.txt")
# ArgoIndex(index_file="bgc-s")  # can use keyword instead of file name: core, bgc-b,␣
↪bgc-b
# ArgoIndex(host="ftp://ftp.ifremer.fr/ifremer/argo")
# ArgoIndex(host="https://data-argo.ifremer.fr", index_file="core")
# ArgoIndex(host="https://data-argo.ifremer.fr", index_file="ar_index_global_prof.txt",␣
↪cache=True)
```

You can then trigger loading of the index content:

```
In [8]: idx.load()  # Load the full index in memory
Out[8]:
<argoindex.pandas>
Host: https://data-argo.ifremer.fr
Index: ar_index_global_prof.txt
Convention: ar_index_global_prof (Profile directory file of the Argo GDAC)
Loaded: True (2963627 records)
Searched: False
```

Here is the list of methods and properties of the **full index**:

```
idx.load(nrows=12)  # Only load the first N rows of the index
idx.N_RECORDS  # Shortcut for length of 1st dimension of the index array
idx.to_dataframe(index=True)  # Convert index to user-friendly :class:`pandas.DataFrame`
idx.to_dataframe(index=True, nrows=2)  # Only returns the first nrows of the index
idx.index  # internal storage structure of the full index (:class:`pyarrow.Table` or :
↪class:`pandas.DataFrame`)
idx.uri_full_index  # List of absolute path to files from the full index table column
↪'file'
```

They are several methods to **search** the index, for instance:

```
In [9]: idx.search_lat_lon_tim([-60, -55, 40., 45., '2007-08-01', '2007-09-01'])
Out[9]:
<argoindex.pandas>
Host: https://data-argo.ifremer.fr
Index: ar_index_global_prof.txt
Convention: ar_index_global_prof (Profile directory file of the Argo GDAC)
Loaded: True (2963627 records)
Searched: True (12 matches, 0.0004%)
```

Here the list of all methods to **search** the index:

```
idx.search_wmo(1901393)
idx.search_cyc(1)
idx.search_wmo_cyc(1901393, [1,12])
idx.search_tim([-60, -55, 40., 45., '2007-08-01', '2007-09-01'])  # Take an index BOX␣
↪definition, only time is used
idx.search_lat_lon([-60, -55, 40., 45., '2007-08-01', '2007-09-01'])  # Take an index␣
↪BOX definition, only lat/lon is used
idx.search_lat_lon_tim([-60, -55, 40., 45., '2007-08-01', '2007-09-01'])  # Take an␣
↪index BOX definition
idx.search_params(['C1PHASE_DOXY', 'DOWNWELLING_PAR'])  # Only for BGC profile index
idx.search_parameter_data_mode({'BBP700': 'D'})  # Only for BGC profile index
```

And finally the list of methods and properties for **search results**:

```
idx.N_MATCH  # Shortcut for length of 1st dimension of the search results array
idx.to_dataframe()  # Convert search results to user-friendly :class:`pandas.DataFrame`
idx.to_dataframe(nrows=2)  # Only returns the first nrows of the search results
idx.to_indexfile("search_index.txt")  # Export search results to Argo standard index file
idx.search  # Internal table with search results
idx.uri  # List of absolute path to files from the search results table column 'file'
```

Usage with bgc index

The **argopy** index store supports the Bio and Synthetic Profile directory files:

```
In [10]: idx = ArgoIndex(index_file="argo_bio-profile_index.txt").load()

# idx = ArgoIndex(index_file="argo_synthetic-profile_index.txt").load()
In [11]: idx
Out[11]:
<argoindex.pandas>
Host: https://data-argo.ifremer.fr
Index: argo_bio-profile_index.txt
Convention: argo_bio-profile_index (Bio-Profile directory file of the Argo GDAC)
Loaded: True (311448 records)
Searched: False
```

Hint: In order to load one BGC-Argo profile index, you can use either `bgc-b` or `bgc-s` keywords to load the `argo_bio-profile_index.txt` or `argo_synthetic-profile_index.txt` index files.

All methods presented *above* are valid with BGC index, but a BGC index store comes with additional search possibilities for parameters and parameter data modes.

Two specific index variables are only available with BGC-Argo index files: `PARAMETERS` and `PARAMETER_DATA_MODE`. We thus implemented the `ArgoIndex.search_params()` and `ArgoIndex.search_parameter_data_mode()` methods. These method allow to search for (i) profiles with one or more specific parameters and (ii) profiles with parameters in one or more specific data modes.

Syntax for `ArgoIndex.search_params()`

1. Load a BGC index

```
In [12]: from argopy import ArgoIndex

In [13]: idx = ArgoIndex(index_file='bgc-s').load()

In [14]: idx
Out[14]:
<argoindex.pandas>
Host: https://data-argo.ifremer.fr
Index: argo_synthetic-profile_index.txt
Convention: argo_synthetic-profile_index (Synthetic-Profile directory file of the Argo␣
↪GDAC)
Loaded: True (310195 records)
Searched: False
```

2. Search for BGC parameters

You can search for one parameter:

```
In [15]: idx.search_params('DOXY')
Out[15]:
<argoindex.pandas>
Host: https://data-argo.ifremer.fr
Index: argo_synthetic-profile_index.txt
Convention: argo_synthetic-profile_index (Synthetic-Profile directory file of the Argo␣
↪GDAC)
Loaded: True (310195 records)
Searched: True (296750 matches, 95.6656%)
```

Or you can search for several parameters:

```
In [16]: idx.search_params(['DOXY', 'CDOM'])
Out[16]:
<argoindex.pandas>
Host: https://data-argo.ifremer.fr
Index: argo_synthetic-profile_index.txt
Convention: argo_synthetic-profile_index (Synthetic-Profile directory file of the Argo␣
↪GDAC)
Loaded: True (310195 records)
Searched: True (48569 matches, 15.6576%)
```

Note that a multiple parameters search will return profiles with *all* parameters. To search for profiles with *any* of the parameters, use:

```
In [17]: idx.search_params(['DOXY', 'CDOM'], logical='or')
Out[17]:
<argoindex.pandas>
Host: https://data-argo.ifremer.fr
Index: argo_synthetic-profile_index.txt
Convention: argo_synthetic-profile_index (Synthetic-Profile directory file of the Argo␣
↪GDAC)
Loaded: True (310195 records)
Searched: True (308944 matches, 99.5967%)
```

Syntax for `ArgoIndex.search_parameter_data_mode()`

1. Load a BGC index

```
In [18]: from argopy import ArgoIndex

In [19]: idx = ArgoIndex(index_file='bgc-b').load()

In [20]: idx
Out[20]:
<argoindex.pandas>
Host: https://data-argo.ifremer.fr
Index: argo_bio-profile_index.txt
```

(continues on next page)

(continued from previous page)

```
Convention: argo_bio-profile_index (Bio-Profile directory file of the Argo GDAC)
Loaded: True (311448 records)
Searched: False
```

2. Search for BGC parameter data mode

You can search one mode for a single parameter:

```
In [21]: idx.search_parameter_data_mode({'BBP700': 'D'})
Out[21]:
<argoindex.pandas>
Host: https://data-argo.ifremer.fr
Index: argo_bio-profile_index.txt
Convention: argo_bio-profile_index (Bio-Profile directory file of the Argo GDAC)
Loaded: True (311448 records)
Searched: True (17529 matches, 5.6282%)
```

You can search several modes for a single parameter:

```
In [22]: idx.search_parameter_data_mode({'DOXY': ['R', 'A']})
Out[22]:
<argoindex.pandas>
Host: https://data-argo.ifremer.fr
Index: argo_bio-profile_index.txt
Convention: argo_bio-profile_index (Bio-Profile directory file of the Argo GDAC)
Loaded: True (311448 records)
Searched: True (104120 matches, 33.4309%)
```

You can search several modes for several parameters:

```
In [23]: idx.search_parameter_data_mode({'BBP700': 'D', 'DOXY': 'D'}, logical='and')
Out[23]:
<argoindex.pandas>
Host: https://data-argo.ifremer.fr
Index: argo_bio-profile_index.txt
Convention: argo_bio-profile_index (Bio-Profile directory file of the Argo GDAC)
Loaded: True (311448 records)
Searched: True (11292 matches, 3.6256%)
```

And mix all of these as you wish:

```
In [24]: idx.search_parameter_data_mode({'BBP700': ['R', 'A'], 'DOXY': 'D'}, logical='or
↪')
Out[24]:
<argoindex.pandas>
Host: https://data-argo.ifremer.fr
Index: argo_bio-profile_index.txt
Convention: argo_bio-profile_index (Bio-Profile directory file of the Argo GDAC)
Loaded: True (311448 records)
Searched: True (234156 matches, 75.1830%)
```

1.8.2 Reference tables

The Argo netcdf format is strict and based on a collection of variables fully documented and conventioned. All reference tables can be found in the Argo user manual.

However, a machine-to-machine access to these tables is often required. This is possible thanks to the work of the **Argo Vocabulary Task Team (AVTT)** that is a team of people responsible for the NVS collections under the Argo Data Management Team governance.

Note: The GitHub organization hosting the AVTT is the 'NERC Vocabulary Server (NVS)', aka 'nvs-vocabs'. This holds a list of NVS collection-specific GitHub repositories. Each Argo GitHub repository is called after its corresponding collection ID (e.g. R01, RR2, R03 etc.). The current list is given here.

The management of issues related to vocabularies managed by the Argo Data Management Team is done on this repository.

---

argopy provides the utility class `ArgoNVSReferenceTables` to easily fetch and get access to all Argo reference tables. If you already know the name of the reference table you want to retrieve, you can simply get it like this:

```
In [25]: from argopy import ArgoNVSReferenceTables

In [26]: NVS = ArgoNVSReferenceTables()

In [27]: NVS.tbl('R01')
Out[27]:
  altLabel  ...                                                id
0    BPROF  ...  http://vocab.nerc.ac.uk/collection/R01/current...
1    BTRAJ  ...  http://vocab.nerc.ac.uk/collection/R01/current...
2     META  ...  http://vocab.nerc.ac.uk/collection/R01/current...
3    MPROF  ...  http://vocab.nerc.ac.uk/collection/R01/current...
4    MTRAJ  ...  http://vocab.nerc.ac.uk/collection/R01/current...
5     PROF  ...  http://vocab.nerc.ac.uk/collection/R01/current...
6    SPROF  ...  http://vocab.nerc.ac.uk/collection/R01/current...
7     TECH  ...  http://vocab.nerc.ac.uk/collection/R01/current...
8     TRAJ  ...  http://vocab.nerc.ac.uk/collection/R01/current...

[9 rows x 5 columns]
```

The reference table is returned as a `pandas.DataFrame`. If you want the exact name of this table:

```
In [28]: NVS.tbl_name('R01')
Out[28]:
('DATA_TYPE',
 'Terms describing the type of data contained in an Argo netCDF file. Argo netCDF␣
↪variable DATA_TYPE is populated by R01 prefLabel.',
 'http://vocab.nerc.ac.uk/collection/R01/current/')
```

If you don't know the reference table ID, you can search for a word in tables title and/or description with the `search` method:

```
In [29]: id_list = NVS.search('sensor')
```

This will return the list of reference table ids matching your search. It can then be used to retrieve table information:

```
In [30]: [NVS.tbl_name(id) for id in id_list]
Out[30]:
[('SENSOR',
  'Terms describing sensor types mounted on Argo floats. Argo netCDF variable SENSOR is␣
↪populated by R25 altLabel.',
  'http://vocab.nerc.ac.uk/collection/R25/current/'),
 ('SENSOR_MAKER',
  'Terms describing developers and manufacturers of sensors mounted on Argo floats. Argo␣
↪netCDF variable SENSOR_MAKER is populated by R26 altLabel.',
  'http://vocab.nerc.ac.uk/collection/R26/current/'),
 ('SENSOR_MODEL',
  'Terms listing models of sensors mounted on Argo floats. Note: avoid using the␣
↪manufacturer name and sensor firmware version in new entries when possible. Argo␣
↪netCDF variable SENSOR_MODEL is populated by R27 altLabel.',
  'http://vocab.nerc.ac.uk/collection/R27/current/')]
```

The full list of all available tables is given by the `ArgoNVSReferenceTables.all_tbl_name()` property. It will return a dictionary with table IDs as key and table name, definition and NVS link as values. Use the `ArgoNVSReferenceTables.all_tbl()` property to retrieve all tables.

```
In [31]: NVS.all_tbl_name
Out[31]:
OrderedDict([('R01',
              ('DATA_TYPE',
               'Terms describing the type of data contained in an Argo netCDF file. Argo␣
↪netCDF variable DATA_TYPE is populated by R01 prefLabel.',
               'http://vocab.nerc.ac.uk/collection/R01/current/')),
             ('R03',
              ('PARAMETER',
               'Terms describing individual measured phenomena, used to mark up sets of␣
↪data in Argo netCDF arrays. Argo netCDF variables PARAMETER and TRAJECTORY_PARAMETERS␣
↪are populated by R03 altLabel; R03 altLabel is also used to name netCDF profile files␣
↪parameter variables <PARAMETER>.',
               'http://vocab.nerc.ac.uk/collection/R03/current/')),
             ('R04',
              ('DATA_CENTRE_CODES',
               'Codes for data centres and institutions handling or managing Argo data.␣
↪Argo netCDF variable DATA_CENTRE is populated by R04 altLabel.',
               'http://vocab.nerc.ac.uk/collection/R04/current/')),
             ('R05',
              ('POSITION_ACCURACY',
               'Accuracy in latitude and longitude measurements received from the␣
↪positioning system, grouped by location accuracy classes.',
               'http://vocab.nerc.ac.uk/collection/R05/current/')),
             ('R06',
              ('DATA_STATE_INDICATOR',
               'Processing stage of the data based on the concatenation of processing␣
↪level and class indicators. Argo netCDF variable DATA_STATE_INDICATOR is populated by␣
↪R06 altLabel.',
               'http://vocab.nerc.ac.uk/collection/R06/current/')),
             ('R07',
              ('HISTORY_ACTION',
               'Coded history information for each action performed on each profile by a␣
```

(continues on next page)

(continued from previous page)

```
↪data centre. Argo netCDF variable HISTORY_ACTION is populated by R07 altLabel.',
               'http://vocab.nerc.ac.uk/collection/R07/current/')),
             ('R08',
              ('ARGO_WMO_INST_TYPE',
               "Subset of instrument type codes from the World Meteorological␣
↪Organization (WMO) Common Code Table C-3 (CCT C-3) 1770, named 'Instrument make and␣
↪type for water temperature profile measurement with fall rate equation coefficients'␣
↪and available here: https://library.wmo.int/doc_num.php?explnum_id=11283. Argo netCDF␣
↪variable WMO_INST_TYPE is populated by R08 altLabel.",
               'http://vocab.nerc.ac.uk/collection/R08/current/')),
             ('R09',
              ('POSITIONING_SYSTEM',
               'List of float location measuring systems. Argo netCDF variable␣
↪POSITIONING_SYSTEM is populated by R09 altLabel.',
               'http://vocab.nerc.ac.uk/collection/R09/current/')),
             ('R10',
              ('TRANS_SYSTEM',
               'List of telecommunication systems. Argo netCDF variable TRANS_SYSTEM is␣
↪populated by R10 altLabel.',
               'http://vocab.nerc.ac.uk/collection/R10/current/')),
             ('R11',
              ('RTQC_TESTID',
               'List of real-time quality-control tests and corresponding binary␣
↪identifiers, used as reference to populate the Argo netCDF HISTORY_QCTEST variable.',
               'http://vocab.nerc.ac.uk/collection/R11/current/')),
             ('R12',
              ('HISTORY_STEP',
               'Data processing step codes for history record. Argo netCDF variable␣
↪TRANS_SYSTEM is populated by R12 altLabel.',
               'http://vocab.nerc.ac.uk/collection/R12/current/')),
             ('R13',
              ('OCEAN_CODE',
               'Ocean area codes assigned to each profile in the Metadata directory␣
↪(index) file of the Argo Global Assembly Centre.',
               'http://vocab.nerc.ac.uk/collection/R13/current/')),
             ('R15',
              ('MEASUREMENT_CODE_ID',
               'Measurement code IDs used in Argo Trajectory netCDF files. Argo netCDF␣
↪variable MEASUREMENT_CODE is populated by R15 altLabel.',
               'http://vocab.nerc.ac.uk/collection/R15/current/')),
             ('R16',
              ('VERTICAL_SAMPLING_SCHEME',
               'Profile sampling schemes and sampling methods. Argo netCDF variable␣
↪VERTICAL_SAMPLING_SCHEME is populated by R16 altLabel.',
               'http://vocab.nerc.ac.uk/collection/R16/current/')),
             ('R18',
              ('CONFIG_PARAMETER_NAME',
               "List of float configuration settings selected by the float Principal␣
↪Investigator (PI). Configuration parameters may or may not be reported by the float,␣
↪and do not constitute float measurements. Configuration parameters selected for a␣
↪float are stored in the float 'meta.nc' file, under CONFIG_PARAMETER_NAME. Each␣
↪configuration parameter name has an associated value, stored in CONFIG_PARAMETER_VALUE.
```

(continues on next page)

(continued from previous page)

```
↪ Argo netCDF variable CONFIG_PARAMETER_NAME is populated by R18 prefLabel.",
               'http://vocab.nerc.ac.uk/collection/R18/current/')),
             ('R19',
              ('STATUS',
               'Flag scale for values in all Argo netCDF cycle timing variables. Argo␣
↪netCDF cycle timing variables JULD_<RTV>_STATUS are populated by R19 altLabel.',
               'http://vocab.nerc.ac.uk/collection/R19/current/')),
             ('R20',
              ('GROUNDED',
               'Codes to indicate the best estimate of whether the float touched the␣
↪ground during a specific cycle. Argo netCDF variable GROUNDED in the Trajectory file␣
↪is populated by R20 altLabel.',
               'http://vocab.nerc.ac.uk/collection/R20/current/')),
             ('R21',
              ('REPRESENTATIVE_PARK_PRESSURE_STATUS',
               'Argo status flag on the Representative Park Pressure (RPP). Argo netCDF␣
↪variable REPRESENTATIVE_PARK_PRESSURE_STATUS in the Trajectory file is populated by␣
↪R21 altLabel.',
               'http://vocab.nerc.ac.uk/collection/R21/current/')),
             ('R22',
              ('PLATFORM_FAMILY',
               'List of platform family/category of Argo floats. Argo netCDF variable␣
↪PLATFORM_FAMILY is populated by R22 altLabel.',
               'http://vocab.nerc.ac.uk/collection/R22/current/')),
             ('R23',
              ('PLATFORM_TYPE',
               'List of Argo float types. Argo netCDF variable PLATFORM_TYPE is␣
↪populated by R23 altLabel.',
               'http://vocab.nerc.ac.uk/collection/R23/current/')),
             ('R24',
              ('PLATFORM_MAKER',
               'List of Argo float manufacturers. Argo netCDF variable PLATFORM_MAKER is␣
↪populated by R24 altLabel.',
               'http://vocab.nerc.ac.uk/collection/R24/current/')),
             ('R25',
              ('SENSOR',
               'Terms describing sensor types mounted on Argo floats. Argo netCDF␣
↪variable SENSOR is populated by R25 altLabel.',
               'http://vocab.nerc.ac.uk/collection/R25/current/')),
             ('R26',
              ('SENSOR_MAKER',
               'Terms describing developers and manufacturers of sensors mounted on Argo␣
↪floats. Argo netCDF variable SENSOR_MAKER is populated by R26 altLabel.',
               'http://vocab.nerc.ac.uk/collection/R26/current/')),
             ('R27',
              ('SENSOR_MODEL',
               'Terms listing models of sensors mounted on Argo floats. Note: avoid␣
↪using the manufacturer name and sensor firmware version in new entries when possible.␣
↪Argo netCDF variable SENSOR_MODEL is populated by R27 altLabel.',
               'http://vocab.nerc.ac.uk/collection/R27/current/')),
             ('R28',
              ('CONTROLLER_BOARD_TYPE',
```

(continues on next page)

(continued from previous page)

```
               'List of Argo floats controller board types and generations. Argo netCDF␣
↪variables CONTROLLER_BOARD_TYPE_PRIMARY and, when needed, CONTROLLER_BOARD_TYPE_
↪SECONDARY, are populated by R28 altLabel.',
               'http://vocab.nerc.ac.uk/collection/R28/current/')),
             ('R40',
              ('PI_NAME',
               'List of Principal Investigator (PI) names in charge of Argo floats. Argo␣
↪netCDF variable PI_NAME is populated by R40 altLabel.',
               'http://vocab.nerc.ac.uk/collection/R40/current/')),
             ('RD2',
              ('DM_QC_FLAG',
               "Quality flag scale for delayed-mode measurements. Argo netCDF variables
↪<PARAMETER>_ADJUSTED_QC in 'D' mode are populated by RD2 altLabel.",
               'http://vocab.nerc.ac.uk/collection/RD2/current/')),
             ('RMC',
              ('MEASUREMENT_CODE_CATEGORY',
               "Categories of trajectory measurement codes listed in NVS collection 'R15'
↪",
               'http://vocab.nerc.ac.uk/collection/RMC/current/')),
             ('RP2',
              ('PROF_QC_FLAG',
               'Quality control flag scale for whole profiles. Argo netCDF variables␣
↪PROFILE_<PARAMETER>_QC are populated by RP2 altLabel.',
               'http://vocab.nerc.ac.uk/collection/RP2/current/')),
             ('RR2',
              ('RT_QC_FLAG',
               "Quality flag scale for real-time measurements. Argo netCDF variables
↪<PARAMETER>_QC in 'R' mode and <PARAMETER>_ADJUSTED_QC in 'A' mode are populated by␣
↪RR2 altLabel.",
               'http://vocab.nerc.ac.uk/collection/RR2/current/')),
             ('RTV',
              ('CYCLE_TIMING_VARIABLE',
               "Timing variables representing stages of an Argo float profiling cycle,␣
↪most of which are associated with a trajectory measurement code ID listed in NVS␣
↪collection 'R15'. Argo netCDF cycle timing variable names JULD_<RTV>_STATUS are␣
↪constructed by RTV altLabel.",
               'http://vocab.nerc.ac.uk/collection/RTV/current/'))])
```

1.8.3 Deployment Plan

It may be useful to be able to retrieve meta-data from Argo deployments. **argopy** can use the OceanOPS API for metadata access to retrieve these information. The returned deployment *plan* is a list of all Argo floats ever deployed, together with their deployment location, date, WMO, program, country, float model and current status.

To fetch the Argo deployment plan, **argopy** provides a dedicated utility class: `OceanOPSDeployments` that can be used like this:

```
In [32]: from argopy import OceanOPSDeployments

In [33]: deployment = OceanOPSDeployments()
```

(continues on next page)

(continued from previous page)

```
In [34]: df = deployment.to_dataframe()
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Cell In[34], line 1
----> 1 df = deployment.to_dataframe()

File ~/checkouts/readthedocs.org/user_builds/argopy/checkouts/latest/argopy/related/
↪ocean_ops_deployments.py:359, in OceanOPSDeployments.to_dataframe(self)
    352 def to_dataframe(self):
    353     """Return the deployment plan as :class:`pandas.DataFrame`
    354
    355     Returns
    356     -------
    357     :class:`pandas.DataFrame`
    358     """
--> 359     data = self.to_json()
    360     if data["total"] == 0:
    361         raise DataNotFound("Your search matches no results")

File ~/checkouts/readthedocs.org/user_builds/argopy/checkouts/latest/argopy/related/
↪ocean_ops_deployments.py:349, in OceanOPSDeployments.to_json(self)
    347 """Return OceanOPS API request response as a json object"""
    348 if self.data is None:
--> 349     self.data = self.fs.open_json(self.uri)
    350 return self.data

File ~/checkouts/readthedocs.org/user_builds/argopy/checkouts/latest/argopy/stores/
↪filesystems.py:1228, in httpstore.open_json(self, url, **kwargs)
   1216 def open_json(self, url, **kwargs):
   1217     """Return a json from an url, or verbose errors
   1218
   1219     Parameters
   (...)
   1226
   1227     """
-> 1228     data = self.download_url(url)
   1229     js = json.loads(data, **kwargs)
   1230     if len(js) == 0:

File ~/checkouts/readthedocs.org/user_builds/argopy/checkouts/latest/argopy/stores/
↪filesystems.py:693, in httpstore.download_url(self, url, n_attempt, max_attempt, cat_
↪opts, *args, **kwargs)
    690     return data, n_attempt
    692 url = self.curateurl(url)
--> 693 data, n = make_request(
    694     self.fs,
    695     url,
    696     n_attempt=n_attempt,
    697     max_attempt=max_attempt,
    698     cat_opts=cat_opts,
    699 )
    701 if data is None:
```

(continues on next page)

(continued from previous page)

```
    702     raise FileNotFoundError(url)

File ~/checkouts/readthedocs.org/user_builds/argopy/checkouts/latest/argopy/stores/
↪filesystems.py:663, in httpstore.download_url.<locals>.make_request(ffs, url, n_
↪attempt, max_attempt, cat_opts)
    661 if n_attempt <= max_attempt:
    662     try:
--> 663         data = ffs.cat_file(url, **cat_opts)
    664     except aiohttp.ClientResponseError as e:
    665         if e.status == 413:

File ~/checkouts/readthedocs.org/user_builds/argopy/envs/latest/lib/python3.8/site-
↪packages/fsspec/asyn.py:118, in sync_wrapper.<locals>.wrapper(*args, **kwargs)
    115 @functools.wraps(func)
    116 def wrapper(*args, **kwargs):
    117     self = obj or args[0]
--> 118     return sync(self.loop, func, *args, **kwargs)

File ~/checkouts/readthedocs.org/user_builds/argopy/envs/latest/lib/python3.8/site-
↪packages/fsspec/asyn.py:103, in sync(loop, func, timeout, *args, **kwargs)
    101     raise FSTimeoutError from return_result
    102 elif isinstance(return_result, BaseException):
--> 103     raise return_result
    104 else:
    105     return return_result

File ~/checkouts/readthedocs.org/user_builds/argopy/envs/latest/lib/python3.8/site-
↪packages/fsspec/asyn.py:56, in _runner(event, coro, result, timeout)
     54     coro = asyncio.wait_for(coro, timeout=timeout)
     55 try:
---> 56     result[0] = await coro
     57 except Exception as ex:
     58     result[0] = ex

File ~/checkouts/readthedocs.org/user_builds/argopy/envs/latest/lib/python3.8/site-
↪packages/fsspec/implementations/http.py:236, in HTTPFileSystem._cat_file(self, url,␣
↪start, end, **kwargs)
    234 async with session.get(self.encode_url(url), **kw) as r:
    235     out = await r.read()
--> 236     self._raise_not_found_for_status(r, url)
    237 return out

File ~/checkouts/readthedocs.org/user_builds/argopy/envs/latest/lib/python3.8/site-
↪packages/fsspec/implementations/http.py:218, in HTTPFileSystem._raise_not_found_for_
↪status(self, response, url)
    214 """
    215 Raises FileNotFoundError for 404s, otherwise uses raise_for_status.
    216 """
    217 if response.status == 404:
--> 218     raise FileNotFoundError(url)
    219 response.raise_for_status()
```

(continues on next page)

(continued from previous page)

```
FileNotFoundError: https://www.ocean-ops.org/api/1/data/platform?exp=[%22networkPtfs.
↪network.name=%27Argo%27%20and%20ptfDepl.deplDate%3E=$var1%22,%20%222024-04-22%2000:00:
↪00%22]&include=%5B%22ref%22,%22ptfDepl.lat%22,%22ptfDepl.lon%22,%22ptfDepl.deplDate%22,
↪%22ptfStatus.id%22,%22ptfStatus.name%22,%22ptfStatus.description%22,%22program.
↪nameShort%22,%22program.country.nameShort%22,%22ptfModel.nameShort%22,%22ptfDepl.noSite
↪%22%5D

In [35]: df
Out[35]:
                                      file  ...                                    ␣
↪profiler
0    nmdis/2901623/profiles/R2901623_000.nc  ...  PROVOR float with SBE conductivity␣
↪sensor
1   nmdis/2901623/profiles/R2901623_000D.nc  ...  PROVOR float with SBE conductivity␣
↪sensor
2    nmdis/2901623/profiles/R2901623_001.nc  ...  PROVOR float with SBE conductivity␣
↪sensor
3    nmdis/2901623/profiles/R2901623_002.nc  ...  PROVOR float with SBE conductivity␣
↪sensor
4    nmdis/2901623/profiles/R2901623_003.nc  ...  PROVOR float with SBE conductivity␣
↪sensor
..                                      ...  ...                                    .
↪..
93   nmdis/2901623/profiles/R2901623_092.nc  ...  PROVOR float with SBE conductivity␣
↪sensor
94   nmdis/2901623/profiles/R2901623_093.nc  ...  PROVOR float with SBE conductivity␣
↪sensor
95   nmdis/2901623/profiles/R2901623_094.nc  ...  PROVOR float with SBE conductivity␣
↪sensor
96   nmdis/2901623/profiles/R2901623_095.nc  ...  PROVOR float with SBE conductivity␣
↪sensor
97   nmdis/2901623/profiles/R2901623_096.nc  ...  PROVOR float with SBE conductivity␣
↪sensor

[98 rows x 12 columns]
```

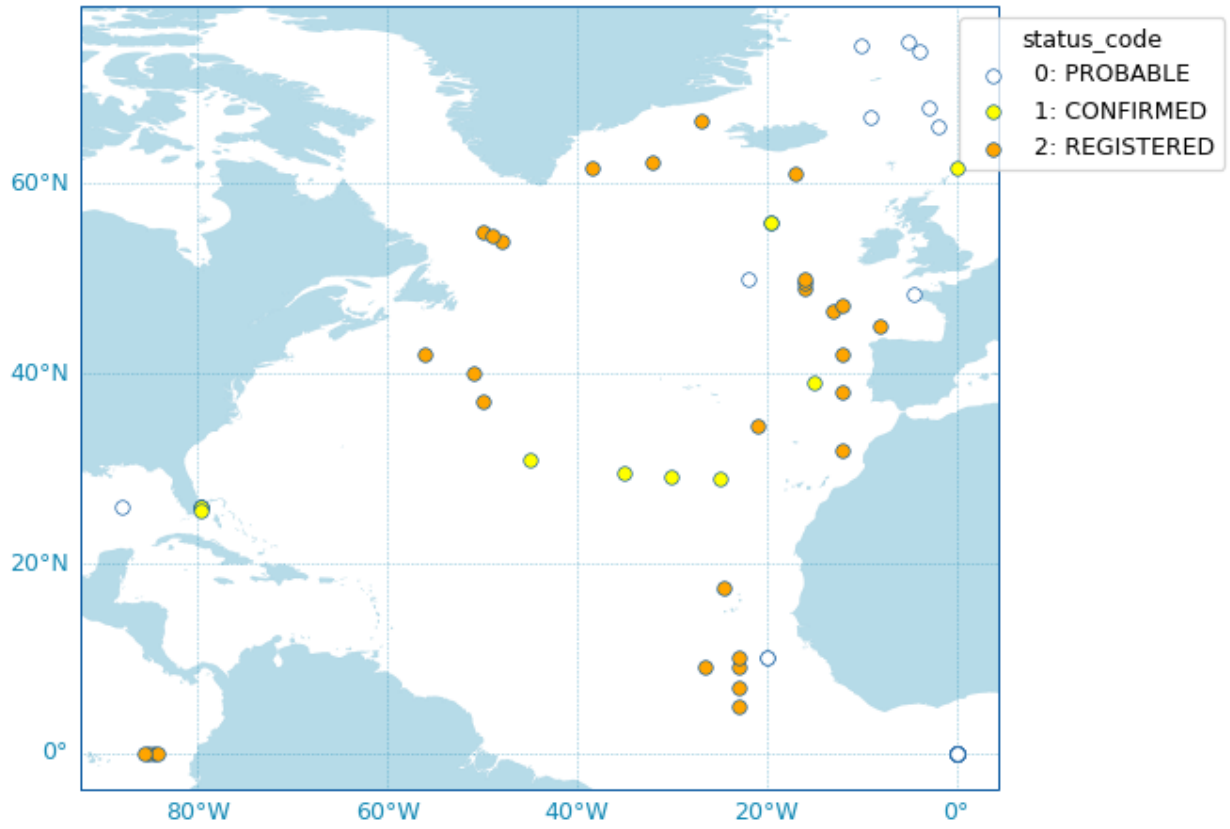
`OceanOPSDeployments` can also take an index box definition as argument in order to restrict the deployment plan selection to a specific region or period:

```
deployment = OceanOPSDeployments([-90, 0, 0, 90])
# deployment = OceanOPSDeployments([-20, 0, 42, 51, '2020-01', '2021-01'])
# deployment = OceanOPSDeployments([-180, 180, -90, 90, '2020-01', None])
```

Note that if the starting date is not provided, it will be set automatically to the current date.

Last, `OceanOPSDeployments` comes with a plotting method:

```
fig, ax = deployment.plot_status()
```



Note: The list of possible deployment status name/code is given by:

```
OceanOPSDeployments().status_code
```

Status	Id	Description
PROBABLE	0	Starting status for some platforms, when there is only a few metadata available, like rough deployment location and date. The platform may be deployed
CONFIRMED	1	Automatically set when a ship is attached to the deployment information. The platform is ready to be deployed, deployment is planned
REGISTERED	2	Starting status for most of the networks, when deployment planning is not done. The deployment is certain, and a notification has been sent via the OceanOPS system
OPERATIONAL	6	Automatically set when the platform is emitting a pulse and observations are distributed within a certain time interval
INACTIVE	4	The platform is not emitting a pulse since a certain time
CLOSED	5	The platform is not emitting a pulse since a long time, it is considered as dead

1.8.4 ADMT Documentation

More than 20 pdf manuals have been produced by the Argo Data Management Team. Using the `ArgoDocs` class, it's easy to navigate this great database.

If you don't know where to start, you can simply list all available documents:

```
In [36]: from argopy import ArgoDocs

In [37]: ArgoDocs().list
Out[37]:
             category  ...     id
0   Argo data formats  ...  29825
1     Quality control  ...  33951
2     Quality control  ...  46542
3     Quality control  ...  40879
4     Quality control  ...  35385
5     Quality control  ...  84370
6     Quality control  ...  62466
7           Cookbooks  ...  41151
8           Cookbooks  ...  29824
9           Cookbooks  ...  78994
10          Cookbooks  ...  39795
11          Cookbooks  ...  39459
12          Cookbooks  ...  39468
13          Cookbooks  ...  47998
14          Cookbooks  ...  54541
15          Cookbooks  ...  46121
16          Cookbooks  ...  51541
17          Cookbooks  ...  57195
18          Cookbooks  ...  46120
19          Cookbooks  ...  52154
20          Cookbooks  ...  55637
21          Cookbooks  ...  46202
22          Cookbooks  ...  57195
23          Cookbooks  ...  46121
24          Cookbooks  ...  57195
25    Quality Control  ...  97828

[26 rows x 4 columns]
```

Or search for a word in the title and/or abstract:

```
In [38]: results = ArgoDocs().search("oxygen")

In [39]: for docid in results:
   ....:     print("\n", ArgoDocs(docid))
   ....:

 <argopy.ArgoDocs>
Title: Argo quality control manual for dissolved oxygen concentration
DOI: 10.13155/46542
url: https://dx.doi.org/10.13155/46542
last pdf: https://archimer.ifremer.fr/doc/00354/46542/82301.pdf
```

(continues on next page)

(continued from previous page)

```
Authors: Thierry, Virginie; Bittig, Henry
Abstract: This document is the Argo quality control manual for Dissolved oxygen␣
↪concentration. It describes two levels of quality control:         - The first level␣
↪is the real-time system that performs a set of agreed automatic checks. Adjustment in␣
↪real-time can also be performed and the real-time system can evaluate quality flags␣
↪for adjusted fields.         - The second level is the delayed-mode quality control␣
↪system.

 <argopy.ArgoDocs>
Title: Processing Argo oxygen data at the DAC level
DOI: 10.13155/39795
url: https://dx.doi.org/10.13155/39795
last pdf: https://archimer.ifremer.fr/doc/00287/39795/94062.pdf
Authors: THIERRY, Virginie; Bittig, Henry; GILBERT, Denis; KOBAYASHI, Taiyo; KANAKO,␣
↪Sato; SCHMID, Claudia
Abstract: This document does NOT address the issue of oxygen data quality control␣
↪(either real-time or delayed mode). As a preliminary step towards that goal, this␣
↪document seeks to ensure that all countries deploying floats equipped with oxygen␣
↪sensors document the data and metadata related to these floats properly. We produced␣
↪this document in response to action item 14 from the AST-10 meeting in Hangzhou (March␣
↪22-23, 2009). Action item 14: Denis Gilbert to work with Taiyo Kobayashi and Virginie␣
↪Thierry to ensure DACs are processing oxygen data according to recommendations. If the␣
↪recommendations contained herein are followed, we will end up with a more uniform set␣
↪of oxygen data within the Argo data system, allowing users to begin analysing not only␣
↪their own oxygen data, but also those of others, in the true spirit of Argo data␣
↪sharing. Indications provided in this document are valid as of the date of writing␣
↪this document. It is very likely that changes in sensors, calibrations and conversions␣
↪equations will occur in the future. Please contact V. Thierry (vthierry@ifremer.fr)␣
↪for any inconsistencies or missing information. A dedicated webpage on the Argo Data␣
↪Management website (www) contains all information regarding Argo oxygen data␣
↪management : current and previous version of this cookbook, oxygen sensor manuals,␣
↪calibration sheet examples, examples of matlab code to process oxygen data, test data,␣
↪etc..
```

Then using the Argo doi number of a document, you can easily retrieve it:

```
In [40]: ArgoDocs(35385)
Out[40]:
<argopy.ArgoDocs>
Title: BGC-Argo quality control manual for the Chlorophyll-A concentration
DOI: 10.13155/35385
url: https://dx.doi.org/10.13155/35385
last pdf: https://archimer.ifremer.fr/doc/00243/35385/60181.pdf
Authors: SCHMECHTIG, Catherine; CLAUSTRE, Herve; POTEAU, Antoine; D'ORTENZIO, Fabrizio;␣
↪Schallenberg, Christina; Trull, Thomas; Xing, Xiaogang
Abstract: This document is the BGC-Argo quality control manual for Chlorophyll A␣
↪concentration. It describes the method used in real-time to apply quality control␣
↪flags to Chlorophyll A concentration calculated from specific sensors mounted on Argo␣
↪profiling floats.
```

and open it in your browser:

```
# ArgoDocs(35385).show()
# ArgoDocs(35385).open_pdf(page=12)
```

1.9 Performances

To improve **argopy** data fetching performances (in terms of time of retrieval), 2 solutions are available:

- *Cache* fetched data, i.e. save your request locally so that you don't have to fetch it again,
- Use *Parallel data fetching*, i.e. fetch chunks of independent data simultaneously.

These solutions are explained below.

Note that another solution from standard big data strategies would be to fetch data lazily. But since (i) **argopy** post-processes raw Argo data on the client side and (ii) none of the data sources are cloud/lazy compatible, this solution is not possible (yet).

Let's start with standard import:

```
In [1]: import argopy

In [2]: from argopy import DataFetcher
```

1.9.1 Cache

Caching data

If you want to avoid retrieving the same data several times during a working session, or if you fetched a large amount of data, you may want to temporarily save data in a cache file.

You can cache fetched data with the fetchers option `cache`.

Argopy cached data are persistent, meaning that they are stored locally on files and will survive execution of your script with a new session. **Cached data have an expiration time of one day**, since this is the update frequency of most data sources. This will ensure you always have the last version of Argo data.

All data and meta-data (index) fetchers have a caching system.

The argopy default cache folder is under your home directory at `~/.cache/argopy`.

But you can specify the path you want to use in several ways:

- with **argopy** global options:

```
argopy.set_options(cachedir='mycache_folder')
```

- in a temporary context:

```
with argopy.set_options(cachedir='mycache_folder'):
    f = DataFetcher(cache=True)
```

- when instantiating the data fetcher:

```
f = DataFetcher(cache=True, cachedir='mycache_folder')
```

Warning: You really need to set the `cache` option to `True`. Specifying only the `cachedir` won't trigger caching !

Clearing the cache

If you want to manually clear your cache folder, and/or make sure your data are newly fetched, you can do it at the fetcher level with the `clear_cache` method.

Start to fetch data and store them in cache:

```
In [3]: argopy.set_options(cachedir='mycache_folder')
Out[3]: <argopy.options.set_options at 0x7fd7128b8610>

In [4]: fetcher1 = DataFetcher(cache=True).profile(6902746, 34).load()
```

Fetched data are in the local cache folder:

```
In [5]: import os

In [6]: os.listdir('mycache_folder')
Out[6]: ['3d6aa407feabc6128d30f54845ff1d78012f0e05a5e791bcf7ed21dedd551a2c', 'cache']
```

where we see hash entries for the newly fetched data and the cache registry file `cache`.

We can then fetch something else using the same cache folder:

```
In [7]: fetcher2 = DataFetcher(cache=True).profile(1901393, 1).load()
```

All fetched data are cached:

```
In [8]: os.listdir('mycache_folder')
Out[8]:
['3d6aa407feabc6128d30f54845ff1d78012f0e05a5e791bcf7ed21dedd551a2c',
 'cache',
 '7f341e5e92d57740746fdfad9eaabae179cd1d845aa926179f009681d3fe86c8']
```

Note the new hash file from *fetcher2* data.

It is important to note that we can safely clear the cache from the first *fetcher1* data without removing *fetcher2* data:

```
In [9]: fetcher1.clear_cache()

In [10]: os.listdir('mycache_folder')
Out[10]: ['cache', '7f341e5e92d57740746fdfad9eaabae179cd1d845aa926179f009681d3fe86c8']
```

By using the fetcher level clear cache, you make sure that only data fetched with it are removed, while other fetched data (with other fetchers for instance) will stay in place.

If you want to clear the entire cache folder, whatever the fetcher used, do it at the package level with:

```
In [11]: argopy.clear_cache()

In [12]: os.listdir('mycache_folder')
Out[12]: []
```

1.9.2 Parallel data fetching

Sometimes you may find that your request takes a long time to fetch, or simply does not even succeed. This is probably because you're trying to fetch a large amount of data.

In this case, you can try to let argopy chunks your request into smaller pieces and have them fetched in parallel for you. This is done with the argument `parallel` of the data fetcher and can be tuned using options `chunks` and `chunksize`.

This goes by default like this:

```
# Define a box to load (large enough to trigger chunking):
In [13]: box = [-60, -30, 40.0, 60.0, 0.0, 100.0, "2007-01-01", "2007-04-01"]

# Instantiate a parallel fetcher:
In [14]: loader_par = DataFetcher(src='erddap', parallel=True).region(box)
```

you can also use the option `progress` to display a progress bar during fetching:

```
In [15]: loader_par = DataFetcher(src='erddap', parallel=True, progress=True).region(box)

In [16]: loader_par
Out[16]:
<datafetcher.erddap>
Name: Ifremer erddap Argo data fetcher for a space/time region
API: https://erddap.ifremer.fr/erddap
Domain: [x=-60.00/-30.00; y=40.00/60.0 ... 00.0; t=2007-01-01/2007-04-01]
Performances: cache=False, parallel=True
User mode: standard
Dataset: phy
```

Then, you can fetch data as usual:

```
In [17]: %%time
   ....: ds = loader_par.to_xarray()
   ....:
Final post-processing of the merged dataset () ...
CPU times: user 742 ms, sys: 20.5 ms, total: 762 ms
Wall time: 7.72 s
```

Number of chunks

To see how many chunks your request has been split into, you can look at the `uri` property of the fetcher, it gives you the list of paths toward data:

```
In [18]: for uri in loader_par.uri:
   ....:     print("http: ... ", "&".join(uri.split("&")[1:-2]))  # Display only the␣
↪relevant part of each URLs of URI:
   ....:
http: ...  longitude>=-60.0&longitude<=-45.0&latitude>=40.0&latitude<=60.0&pres>=0.0&pres
↪<=100.0&time>=1167609600.0&time<=1175385600.0
http: ...  longitude>=-45.0&longitude<=-30.0&latitude>=40.0&latitude<=60.0&pres>=0.0&pres
↪<=100.0&time>=1167609600.0&time<=1175385600.0
```

To control chunking, you can use the ``**chunks**`` option that specifies the number of chunks in each of the *direction*:

- `lon`, `lat`, `dpt` and `time` for a **region** fetching,
- `wmo` for a **float** and **profile** fetching.

```
# Create a large box:
In [19]: box = [-60, 0, 0.0, 60.0, 0.0, 500.0, "2007", "2010"]

# Init a parallel fetcher:
In [20]: loader_par = DataFetcher(src='erddap',
   ....:                                parallel=True,
   ....:                                chunks={'lon': 5}).region(box)
   ....:

# Check number of chunks:
In [21]: len(loader_par.uri)
Out[21]: 195
```

This creates 195 chunks, and 5 along the longitudinale direction, as requested.

When the `chunks` option is not specified for a given *direction*, it relies on auto-chunking using pre-defined chunk maximum sizes (see below). In the case above, auto-chunking appends also along latitude, depth and time; this explains why we have 195 and not only 5 chunks.

To chunk the request along a single direction, set explicitly all the other directions to 1:

```
# Init a parallel fetcher:
In [22]: loader_par = DataFetcher(src='erddap',
   ....:                                parallel=True,
   ....:                                chunks={'lon': 5, 'lat':1, 'dpt':1, 'time':1}).
↪region(box)
   ....:

# Check number of chunks:
In [23]: len(loader_par.uri)
Out[23]: 5
```

We now have 5 chunks along longitude, check out the URLs parameter in the list of URIs:

```
In [24]: for uri in loader_par.uri:
   ....:     print("&".join(uri.split("&")[1:-2])) # Display only the relevant URL part
```

(continues on next page)

(continued from previous page)

```
   ....:
longitude>=-60.0&longitude<=-48.0&latitude>=0.0&latitude<=60.0&pres>=0.0&pres<=500.0&
↪time>=1167609600.0&time<=1262304000.0
longitude>=-48.0&longitude<=-36.0&latitude>=0.0&latitude<=60.0&pres>=0.0&pres<=500.0&
↪time>=1167609600.0&time<=1262304000.0
longitude>=-36.0&longitude<=-24.0&latitude>=0.0&latitude<=60.0&pres>=0.0&pres<=500.0&
↪time>=1167609600.0&time<=1262304000.0
longitude>=-24.0&longitude<=-12.0&latitude>=0.0&latitude<=60.0&pres>=0.0&pres<=500.0&
↪time>=1167609600.0&time<=1262304000.0
longitude>=-12.0&longitude<=0.0&latitude>=0.0&latitude<=60.0&pres>=0.0&pres<=500.0&time>
↪=1167609600.0&time<=1262304000.0
```

Note: You may notice that if you run the last command with the *argovis* fetcher, you will still have more than 5 chunks (i.e. 65). This is because *argovis* is limited to 3 months length requests. So, for this request that is 3 years long, argopy ends up with 13 chunks along time, times 5 chunks in longitude, leading to 65 chunks in total.

Warning: The *gdac* fetcher and the *float* and *profile* access points of the *argovis* fetcher use a list of resources than are not chunked but fetched in parallel using a batch queue.

Size of chunks

The default chunk size for each access point dimensions are:

Access point dimension	Maximum chunk size
region / **lon**	20 deg
region / **lat**	20 deg
region / **dpt**	500 m or db
region / **time**	90 days
float / **wmo**	5
profile / **wmo**	5

These default values are used to chunk data when the `chunks` parameter key is set to `auto`.

But you can modify the maximum chunk size allowed in each of the possible directions. This is done with the option ``**chunks_maxsize**``.

For instance if you want to make sure that your chunks are not larger then 100 meters (db) in depth (pressure), you can use:

```
# Create a large box:
In [25]: box = [-60, -10, 40.0, 60.0, 0.0, 500.0, "2007", "2010"]

# Init a parallel fetcher:
In [26]: loader_par = DataFetcher(src='erddap',
   ....:                          parallel=True,
   ....:                          chunks_maxsize={'dpt': 100}).region(box)
   ....:
```

(continues on next page)

(continued from previous page)

```
# Check number of chunks:
In [27]: len(loader_par.uri)
Out[27]: 195
```

Since this creates a large number of chunks, let's do this again and combine with the option `chunks` to see easily what's going on:

```
# Init a parallel fetcher with chunking along the vertical axis alone:
In [28]: loader_par = DataFetcher(src='erddap',
   ....:                                parallel=True,
   ....:                                chunks_maxsize={'dpt': 100},
   ....:                                chunks={'lon':1, 'lat':1, 'dpt':'auto', 'time':1}).
↪region(box)
   ....:

In [29]: for uri in loader_par.uri:
   ....:     print("http: ... ", "&".join(uri.split("&")[1:-2])) # Display only the␣
↪relevant URL part
   ....:
http: ...  longitude>=-60&longitude<=-10&latitude>=40.0&latitude<=60.0&pres>=0.0&pres
↪<=100.0&time>=1167609600.0&time<=1262304000.0
http: ...  longitude>=-60&longitude<=-10&latitude>=40.0&latitude<=60.0&pres>=100.0&pres
↪<=200.0&time>=1167609600.0&time<=1262304000.0
http: ...  longitude>=-60&longitude<=-10&latitude>=40.0&latitude<=60.0&pres>=200.0&pres
↪<=300.0&time>=1167609600.0&time<=1262304000.0
http: ...  longitude>=-60&longitude<=-10&latitude>=40.0&latitude<=60.0&pres>=300.0&pres
↪<=400.0&time>=1167609600.0&time<=1262304000.0
http: ...  longitude>=-60&longitude<=-10&latitude>=40.0&latitude<=60.0&pres>=400.0&pres
↪<=500.0&time>=1167609600.0&time<=1262304000.0
```

You can see, that the `pres` argument of this erddap list of URLs define layers not thicker than the requested 100db.

With the `profile` and `float` access points, you can use the `wmo` keyword to control the number of WMOs in each chunks.

```
In [30]: WMO_list = [6902766, 6902772, 6902914, 6902746, 6902916, 6902915, 6902757,␣
↪6902771]

# Init a parallel fetcher with chunking along the list of WMOs:
In [31]: loader_par = DataFetcher(src='erddap',
   ....:                                parallel=True,
   ....:                                chunks_maxsize={'wmo': 3}).float(WMO_list)
   ....:

In [32]: for uri in loader_par.uri:
   ....:     print("http: ... ", "&".join(uri.split("&")[1:-2])) # Display only the␣
↪relevant URL part
   ....:
http: ...  platform_number=~"6902766|6902772|6902914"
http: ...  platform_number=~"6902746|6902916|6902915"
http: ...  platform_number=~"6902757|6902771"
```

You see here, that this request for 8 floats is split in chunks with no more that 3 floats each.

Warning: At this point, there is no mechanism to chunk requests along cycle numbers for the `profile` access point.

Parallelization methods

They are 2 methods available to set-up your data fetching requests in parallel:

1. Multi-threading for all data sources,
2. Multi-processing for *gdac* with a local host.

Both options use a pool of threads or processes managed with the concurrent futures module.

The parallelization method is set with the `parallel_method` option of the fetcher, which can take as values `thread` or `process`.

Methods available for data sources:

Parallel method	erddap	gdac	argovis
Multi-threading	X	X	X
Multi-processes		X	

Note that you can in fact pass the method directly with the `parallel` option, so that in practice, the following two formulations are equivalent:

```
In [33]: DataFetcher(parallel=True, parallel_method='thread')
Out[33]:
<datafetcher.erddap> 'No access point initialised'
Available access points: float, profile, region
Performances: cache=False, parallel=True
User mode: standard
Dataset: phy

In [34]: DataFetcher(parallel='thread')
Out[34]:
<datafetcher.erddap> 'No access point initialised'
Available access points: float, profile, region
Performances: cache=False, parallel=thread
User mode: standard
Dataset: phy
```

Comparison of performances

Note that to compare performances with or without the parallel option, we need to make sure that data are not cached on the server side. To do this, we use a very small random perturbation on the box definition, here on the maximum latitude. This ensures that nearly the same amount of data will be requested but not cached by the server.

```
In [35]: def this_box():
   ....:     return [-60, 0,
   ....:             20.0, 60.0 + np.random.randint(0,100,1)[0]/1000,
   ....:             0.0, 500.0,
```

(continues on next page)

(continued from previous page)

```
   ....:             "2007", "2009"]
   ....:
```

```
In [36]: %%time
   ....: b1 = this_box()
   ....: f1 = DataFetcher(src='argovis', parallel=False).region(b1)
   ....: ds1 = f1.to_xarray()
   ....:
Error: 502, message='Proxy Error', url=URL('https://argovisbeta02.colorado.edu/selection/
↪profiles?startDate=2008-07-22T13:20:00Z&endDate=2008-10-11T18:40:00Z&shape=%5B%5B%5B-
↪60,20.0%5D,%5B-60,60.05%5D,%5B0,60.05%5D,%5B0,20.0%5D,%5B-60,20.0%5D%5D%5D&presRange=
↪%5B0.0,500.0%5D')
CPU times: user 7.03 s, sys: 189 ms, total: 7.22 s
Wall time: 4min 20s
```

```
In [37]: %%time
   ....: b2 = this_box()
   ....: f2 = DataFetcher(src='argovis', parallel=True).region(b2)
   ....: ds2 = f2.to_xarray()
   ....:
CPU times: user 8.74 s, sys: 358 ms, total: 9.1 s
Wall time: 36.6 s
```

This simple comparison hopefully shows that parallel request is significantly faster than the standard one.

Warnings

- Parallelizing your fetcher is useful to handle large region of data, but it can also add a significant overhead on *reasonable* size requests that may lead to degraded performances. So, we do not recommend for you to use the parallel option systematically.
- You may have different dataset sizes with and without the `parallel` option. This may happen if one of the chunk data fetching fails. By default, data fetching of multiple resources fails with a warning. You can change this behaviour with the option `errors` of the `to_xarray()` fetcher methods, just set it to `raise` like this:

```
DataFetcher(parallel=True).region(this_box()).to_xarray(errors='raise');
```

You can also use `silent` to simply hide all messages during fetching.

Help & reference

- *What's New*
- *Contributing to argopy*
- *API reference*

1.10 What's New

1.10.1 Coming up next

Internals

- Fix bug with ArgoIndex cache, #345. (#346) by G. Maze.
- Update `argopy.ArgoDocs` with last BGC cookbooks on pH. (#321) by G. Maze.
- Fix for fsspec > 2023.10.0. (#318) by G. Maze.
- Keep dependencies up to date. (#333, #337) by G. Maze.

1.10.2 v0.1.15 (12 Dec. 2023)

Internals

- Fix bug whereby user name could not be retrieved using `getpass.getuser()`. This closes #310 and allows argopy to be integrated into the EU Galaxy tools for ecology. (#311) by G. Maze.

1.10.3 v0.1.14 (29 Sep. 2023)

New in version v0.1.14: This new release brings to pip and conda default install of argopy all new features introduced in the release candidate v0.1.14rc2 and v0.1.14rc1. For simplicity we merged all novelties to this v0.1.14 changelog.

Features and front-end API

- **argopy now support BGC dataset in `expert` user mode for the `erddap` data source**. The BGC-Argo content of synthetic multi-profile files is now available from the Ifremer erddap. Like for the core dataset, you can fetch data for a region, float(s) or profile(s). One novelty with regard to core, is that you can restrict data fetching to some parameters and furthermore impose no-NaNs on some of these parameters. Check out the new documentation page for *Dataset*. (#278) by G. Maze

```
import argopy
from argopy import DataFetcher

argopy.set_options(src='erddap', mode='expert')

DataFetcher(ds='bgc')  # All variables found in the access point will be returned
DataFetcher(ds='bgc', params='all')  # Default: All variables found in the access point␣
↪will be returned
DataFetcher(ds='bgc', params='DOXY') # Only the DOXY variable will be returned
DataFetcher(ds='bgc', params=['DOXY', 'BBP700']) # Only DOXY and BBP700 will be returned

DataFetcher(ds='bgc', measured=None)  # Default: all params are allowed to have NaNs
DataFetcher(ds='bgc', measured='all')  # All params found in the access point cannot be␣
↪NaNs
DataFetcher(ds='bgc', measured='DOXY') # Only DOXY cannot be NaNs
DataFetcher(ds='bgc', measured=['DOXY', 'BBP700']) # Only DOXY and BBP700 cannot be NaNs
```

(continues on next page)

(continued from previous page)

```
DataFetcher(ds='bgc', params='all', measured=None)  # Return the largest possible dataset
DataFetcher(ds='bgc', params='all', measured='all')  # Return the smallest possible␣
↪dataset
DataFetcher(ds='bgc', params='all', measured=['DOXY', 'BBP700'])  # Return all possible␣
↪params for points where DOXY and BBP700 are not NaN
```

- **New methods in the ArgoIndex for BGC**. The *ArgoIndex* has now full support for the BGC profile index files, both bio and synthetic index. In particular it is possible to search for profiles with specific data modes on parameters. (#278) by G. Maze

```
from argopy import ArgoIndex

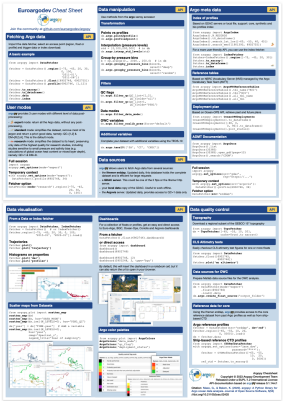
idx = ArgoIndex(index_file="bgc-b")  # Use keywords instead of exact file names: `core`,␣
↪`bgc-b`, `bgc-s`
idx.search_params(['C1PHASE_DOXY', 'DOWNWELLING_PAR'])  # Search for profiles with␣
↪parameters
idx.search_parameter_data_mode({'TEMP': 'D'})  # Search for profiles with specific data␣
↪modes
idx.search_parameter_data_mode({'BBP700': 'D'})
idx.search_parameter_data_mode({'DOXY': ['R', 'A']})
idx.search_parameter_data_mode({'DOXY': 'D', 'CDOM': 'D'}, logical='or')
```

- **New xarray argo accessor features**. Easily retrieve an Argo sample index and domain extent with the `index` and `domain` properties. Get a list with all possible (PLATFORM_NUMBER, CYCLE_NUMBER) with the `list_WMO_CYC` method. (#278) by G. Maze
- **New search methods for Argo reference tables**. It is now possible to search for a string in tables title and/or description using the `related.ArgoNVSReferenceTables.search()` method.

```
from argopy import ArgoNVSReferenceTables

id_list = ArgoNVSReferenceTables().search('sensor')
```

- **Updated documentation**. In order to better introduce new features, we updated the documentation structure and content.
- **argopy** cheatsheet ! Get most of the argopy API in a 2 pages pdf !



- **Our internal Argo index store is promoted as a frontend feature**. The *IndexFetcher* is a user-friendly **fetcher** built on top of our internal Argo index file store. But if you are familiar with Argo index files and/or

cares about performances, you may be interested in using directly the Argo index **store**. We thus decided to promote this internal feature as a frontend class `ArgoIndex`. See *Store: Low-level Argo Index access*. (#270) by G. Maze

- **Easy access to all Argo manuals from the ADMT**. More than 20 pdf manuals have been produced by the Argo Data Management Team. Using the new `ArgoDocs` class, it's now easier to navigate this great database for Argo experts. All details in *ADMT Documentation*. (#268) by G. Maze

```
from argopy import ArgoDocs

ArgoDocs().list

ArgoDocs(35385)
ArgoDocs(35385).ris
ArgoDocs(35385).abstract
ArgoDocs(35385).show()
ArgoDocs(35385).open_pdf()
ArgoDocs(35385).open_pdf(page=12)

ArgoDocs().search("CDOM")
```

- **New 'research' user mode**. This new feature implements automatic filtering of Argo data following international recommendations for research/climate studies. With this user mode, only Delayed Mode with good QC data are returned. Check out the *User mode (, ,)* section for all the details. (#265) by G. Maze
- **argopy now provides a specific xarray engine to properly read Argo netcdf files**. Using `engine='argo'` in `xarray.open_dataset()`, all variables will properly be casted, i.e. returned with their expected data types, which is not the case otherwise. This works with *ALL* Argo netcdf file types (as listed in the Reference table R01). Some details in here: `argopy.xarray.ArgoEngine` (#208) by G. Maze

```
import xarray as xr
ds = xr.open_dataset("dac/aoml/1901393/1901393_prof.nc", engine='argo')
```

- **argopy now can provide authenticated access to the Argo CTD reference database for DMQC**. Using user/password new **argopy** options, it is possible to fetch the Argo CTD reference database, with the `CTDRefDataFetcher` class. (#256) by G. Maze

```
from argopy import CTDRefDataFetcher

with argopy.set_options(user="john_doe", password="***"):
    f = CTDRefDataFetcher(box=[15, 30, -70, -60, 0, 5000.0])
    ds = f.to_xarray()
```

Warning: **argopy** is ready but the Argo CTD reference database for DMQC is not fully published on the Ifremer ERDDAP yet. This new feature will thus be fully operational soon, and while it's not, **argopy** should raise an `ErddapHTTPNotFound` error when using the new fetcher.

- New option to control the expiration time of cache file: `cache_expiration`.

Internals

- Utilities refactoring. All classes and functions have been refactored to more appropriate locations like `argopy.utils` or `argopy.related`. A deprecation warning message should be displayed every time utilities are being used from the deprecated locations. (#290) by G. Maze
- Fix bugs due to fsspec new internal cache handling and Windows specifics. (#293) by G. Maze

- New utility class `utils.MonitoredThreadPoolExecutor` to handle parallelization with a multi-threading Pool that provide a notebook or terminal computation progress dashboard. This class is used by the httpstore open_mfdataset method for erddap requests.
- New utilites to handle a collection of datasets: *utils.drop_variables_not_in_all_datasets()* will drop variables that are not in all datasets (the lowest common denominator) and *utils.fill_variables_not_in_all_datasets()* will add empty variables to dataset so that all the collection have the same data_vars and coords. These functions are used by stores to concat/merge a collection of datasets (chunks).
- `related.load_dict()` now relies on *ArgoNVSReferenceTables* instead of static pickle files.
- *argopy.ArgoColors* colormap for Argo Data-Mode has now a fourth value to account for a white space Fill-Value.
- New quick and dirty plot method *plot.scatter_plot()*
- Refactor pickle files in argopy/assets as json files in argopy/static/assets
- Refactor list of variables by data types used in `related.cast_Argo_variable_type()` into assets json files in argopy/static/assets
- Change of behaviour: when setting the cachedir option, path it's not tested for existence but for being writable, and is created if doesn't exists (but seems to break CI upstream in Windows)
- And misc. bug and warning fixes all over the code.
- Update new argovis dashboard links for floats and profiles. (#271) by G. Maze
- **Index store can now export search results to standard Argo index file format**. See all details in *Store: Low-level Argo Index access*. (#260) by G. Maze

```
from argopy import ArgoIndex as indexstore
# or:
# from argopy.stores import indexstore_pd as indexstore
# or:
# from argopy.stores import indexstore_pa as indexstore

idx = indexstore().search_wmo(3902131)  # Perform any search
idx.to_indexfile('short_index.txt')  # export search results as standard Argo index csv␣
↪file
```

- **Index store can now load/search the Argo Bio and Synthetic profile index files**. Simply gives the name of the Bio or Synthetic Profile index file and retrieve the full index. This store also comes with a new search criteria for BGC: by parameters. See all details in *Store: Low-level Argo Index access*. (#261) by G. Maze

```
from argopy import ArgoIndex as indexstore
# or:
# from argopy.stores import indexstore_pd as indexstore
# or:
# from argopy.stores import indexstore_pa as indexstore

idx = indexstore(index_file="argo_bio-profile_index.txt").load()
idx.search_params(['C1PHASE_DOXY', 'DOWNWELLING_PAR'])
```

- Use a mocked server for all http and GDAC ftp requests in CI tests (#249, #252, #255) by G. Maze
- Removed support for minimal dependency requirements and for python 3.7. (#252) by G. Maze
- Changed License from Apache to EUPL 1.2

Breaking changes

- Some documentation pages may have moved to new urls.
- The legacy index store is deprecated, now available in argopy.stores.argo_index_deprec.py only (#270) by G. Maze

1.10.4 v0.1.14rc2 (27 Jul. 2023)

Features and front-end API

- **argopy now support BGC dataset in `expert` user mode for the `erddap` data source**. The BGC-Argo content of synthetic multi-profile files is now available from the Ifremer erddap. Like for the core dataset, you can fetch data for a region, float(s) or profile(s). One novelty with regard to core, is that you can restrict data fetching to some parameters and furthermore impose no-NaNs on some of these parameters. Check out the new documentation page for *Dataset*. (#278) by G. Maze

```
import argopy
from argopy import DataFetcher

argopy.set_options(src='erddap', mode='expert')

DataFetcher(ds='bgc')  # All variables found in the access point will be returned
DataFetcher(ds='bgc', params='all')  # Default: All variables found in the access point␣
↪will be returned
DataFetcher(ds='bgc', params='DOXY') # Only the DOXY variable will be returned
DataFetcher(ds='bgc', params=['DOXY', 'BBP700']) # Only DOXY and BBP700 will be returned

DataFetcher(ds='bgc', measured=None)  # Default: all params are allowed to have NaNs
DataFetcher(ds='bgc', measured='all')  # All params found in the access point cannot be␣
↪NaNs
DataFetcher(ds='bgc', measured='DOXY') # Only DOXY cannot be NaNs
DataFetcher(ds='bgc', measured=['DOXY', 'BBP700']) # Only DOXY and BBP700 cannot be NaNs

DataFetcher(ds='bgc', params='all', measured=None)  # Return the largest possible dataset
DataFetcher(ds='bgc', params='all', measured='all')  # Return the smallest possible␣
↪dataset
DataFetcher(ds='bgc', params='all', measured=['DOXY', 'BBP700'])  # Return all possible␣
↪params for points where DOXY and BBP700 are not NaN
```

- **New methods in the ArgoIndex for BGC**. The *ArgoIndex* has now full support for the BGC profile index files, both bio and synthetic index. In particular it is possible to search for profiles with specific data modes on parameters. (#278) by G. Maze

```
from argopy import ArgoIndex

idx = ArgoIndex(index_file="bgc-b")  # Use keywords instead of exact file names: `core`,␣
↪`bgc-b`, `bgc-s`
idx.search_params(['C1PHASE_DOXY', 'DOWNWELLING_PAR'])  # Search for profiles with␣
↪parameters
idx.search_parameter_data_mode({'TEMP': 'D'})  # Search for profiles with specific data␣
↪modes
idx.search_parameter_data_mode({'BBP700': 'D'})
idx.search_parameter_data_mode({'DOXY': ['R', 'A']})
idx.search_parameter_data_mode({'DOXY': 'D', 'CDOM': 'D'}, logical='or')
```

- **New xarray argo accessor features**. Easily retrieve an Argo sample index and domain extent with the `index` and `domain` properties. Get a list with all possible (PLATFORM_NUMBER, CYCLE_NUMBER) with the `list_WMO_CYC` method. (#278) by G. Maze
- **New search methods for Argo reference tables**. It is now possible to search for a string in tables title and/or description using the `related.ArgoNVSReferenceTables.search()` method.

```
from argopy import ArgoNVSReferenceTables

id_list = ArgoNVSReferenceTables().search('sensor')
```

- **Updated documentation**. In order to better introduce new features, we updated the documentation structure and content.

Internals

- New utility class `utils.MonitoredThreadPoolExecutor` to handle parallelization with a multi-threading Pool that provide a notebook or terminal computation progress dashboard. This class is used by the httpstore open_mfdataset method for erddap requests.
- New utilites to handle a collection of datasets: *utils.drop_variables_not_in_all_datasets()* will drop variables that are not in all datasets (the lowest common denominator) and *utils.fill_variables_not_in_all_datasets()* will add empty variables to dataset so that all the collection have the same data_vars and coords. These functions are used by stores to concat/merge a collection of datasets (chunks).
- `related.load_dict()` now relies on *ArgoNVSReferenceTables* instead of static pickle files.
- *argopy.ArgoColors* colormap for Argo Data-Mode has now a fourth value to account for a white space Fill-Value.
- New quick and dirty plot method *plot.scatter_plot()*
- Refactor pickle files in argopy/assets as json files in argopy/static/assets
- Refactor list of variables by data types used in `related.cast_Argo_variable_type()` into assets json files in argopy/static/assets
- Change of behaviour: when setting the cachedir option, path it's not tested for existence but for being writable, and is created if doesn't exists (but seems to break CI upstream in Windows)
- And misc. bug and warning fixes all over the code.

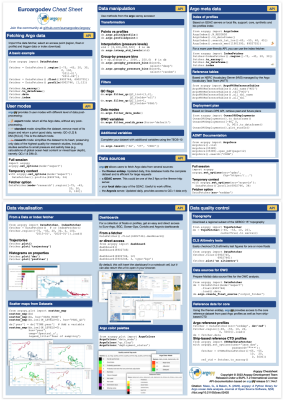
Breaking changes

- Some documentation pages may have moved to new urls.

1.10.5 v0.1.14rc1 (31 May 2023)

Features and front-end API

- **argopy** cheatsheet ! Get most of the argopy API in a 2 pages pdf !



- **Our internal Argo index store is promoted as a frontend feature**. The `IndexFetcher` is a user-friendly **fetcher** built on top of our internal Argo index file store. But if you are familiar with Argo index files and/or cares about performances, you may be interested in using directly the Argo index **store**. We thus decided to promote this internal feature as a frontend class `ArgoIndex`. See *Store: Low-level Argo Index access*. (#270) by G. Maze

- **Easy access to all Argo manuals from the ADMT**. More than 20 pdf manuals have been produced by the Argo Data Management Team. Using the new `ArgoDocs` class, it's now easier to navigate this great database for Argo experts. All details in *ADMT Documentation*. (#268) by G. Maze

```
from argopy import ArgoDocs

ArgoDocs().list

ArgoDocs(35385)
ArgoDocs(35385).ris
ArgoDocs(35385).abstract
ArgoDocs(35385).show()
ArgoDocs(35385).open_pdf()
ArgoDocs(35385).open_pdf(page=12)

ArgoDocs().search("CDOM")
```

- **New 'research' user mode**. This new feature implements automatic filtering of Argo data following international recommendations for research/climate studies. With this user mode, only Delayed Mode with good QC data are returned. Check out the *User mode (, ,)* section for all the details. (#265) by G. Maze

- **argopy now provides a specific xarray engine to properly read Argo netcdf files**. Using `engine='argo'` in `xarray.open_dataset()`, all variables will properly be casted, i.e. returned with their expected data types, which is not the case otherwise. This works with *ALL* Argo netcdf file types (as listed in the Reference table R01). Some details in here: `argopy.xarray.ArgoEngine` (#208) by G. Maze

```
import xarray as xr
ds = xr.open_dataset("dac/aoml/1901393/1901393_prof.nc", engine='argo')
```

- **argopy now can provide authenticated access to the Argo CTD reference database for DMQC**. Using user/password new **argopy** options, it is possible to fetch the Argo CTD reference database, with the `CTDRefDataFetcher` class. (#256) by G. Maze

```
from argopy import CTDRefDataFetcher

with argopy.set_options(user="john_doe", password="***"):
```

(continues on next page)

(continued from previous page)

```
    f = CTDRefDataFetcher(box=[15, 30, -70, -60, 0, 5000.0])
    ds = f.to_xarray()
```

Warning: **argopy** is ready but the Argo CTD reference database for DMQC is not fully published on the Ifremer ERDDAP yet. This new feature will thus be fully operational soon, and while it's not, **argopy** should raise an `ErddapHTTPNotFound` error when using the new fetcher.

- New option to control the expiration time of cache file: `cache_expiration`.

Internals

- Update new argovis dashboard links for floats and profiles. (#271) by G. Maze
- **Index store can now export search results to standard Argo index file format**. See all details in *Store: Low-level Argo Index access*. (#260) by G. Maze

```
from argopy import ArgoIndex as indexstore
# or:
# from argopy.stores import indexstore_pd as indexstore
# or:
# from argopy.stores import indexstore_pa as indexstore

idx = indexstore().search_wmo(3902131)  # Perform any search
idx.to_indexfile('short_index.txt')  # export search results as standard Argo index csv␣
↪file
```

- **Index store can now load/search the Argo Bio and Synthetic profile index files**. Simply gives the name of the Bio or Synthetic Profile index file and retrieve the full index. This store also comes with a new search criteria for BGC: by parameters. See all details in *Store: Low-level Argo Index access*. (#261) by G. Maze

```
from argopy import ArgoIndex as indexstore
# or:
# from argopy.stores import indexstore_pd as indexstore
# or:
# from argopy.stores import indexstore_pa as indexstore

idx = indexstore(index_file="argo_bio-profile_index.txt").load()
idx.search_params(['C1PHASE_DOXY', 'DOWNWELLING_PAR'])
```

- Use a mocked server for all http and GDAC ftp requests in CI tests (#249, #252, #255) by G. Maze
- Removed support for minimal dependency requirements and for python 3.7. (#252) by G. Maze
- Changed License from Apache to EUPL 1.2

Breaking changes

- The legacy index store is deprecated, now available in argopy.stores.argo_index_deprec.py only (#270) by G. Maze

1.10.6 v0.1.13 (28 Mar. 2023)

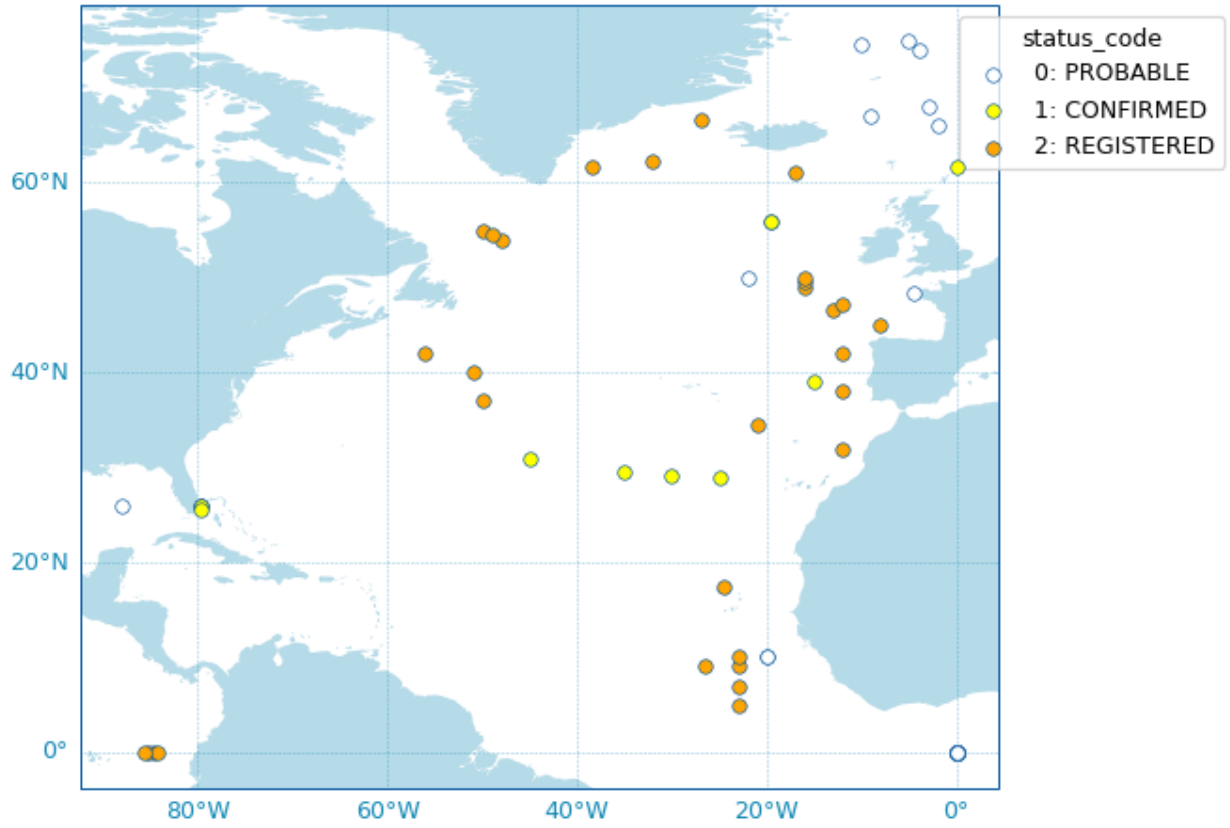
Features and front-end API

- **New utility class to retrieve the Argo deployment plan from the Ocean-OPS api.** This is the utility class `OceanOPSDeployments`. See the new documentation section on *Deployment Plan* for more. (#244) by G. Maze

```
from argopy import OceanOPSDeployments

deployment = OceanOPSDeployments()
deployment = OceanOPSDeployments([-90,0,0,90])
deployment = OceanOPSDeployments([-90,0,0,90], deployed_only=True) # Remove planification

df = deployment.to_dataframe()
deployment.status_code
fig, ax = deployment.plot_status()
```



- **New scatter map utility for easy Argo-related variables plotting.** The new `argopy.plot.scatter_map()` utility function is dedicated to making maps with Argo profiles positions coloured according to specific variables: a scatter map. Profiles colouring is finely tuned for some variables: QC flags, Data Mode and Deployment Status. By default, floats trajectories are always shown, but this can be changed. See the new documentation section on *Scatter Maps* for more. (#245) by G. Maze

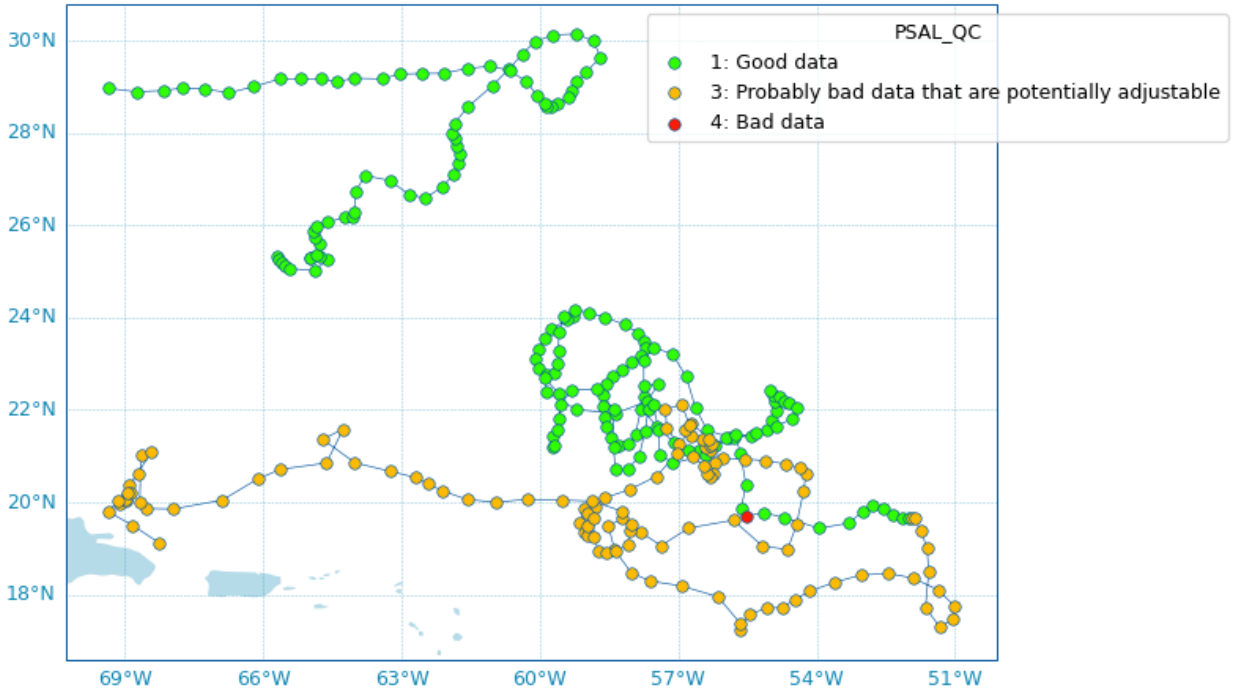
```
from argopy.plot import scatter_map

fig, ax = scatter_map(ds_or_df,
                      x='LONGITUDE', y='LATITUDE', hue='PSAL_QC',
```

(continues on next page)

(continued from previous page)

```
                    traj_axis='PLATFORM_NUMBER')
```



- **New Argo colors utility to manage segmented colormaps and pre-defined Argo colors set.** The new `argopy.plot.ArgoColors` utility class aims to easily provide colors for Argo-related variables plot. See the new documentation section on *Argo colors* for more (#245) by G. Maze

```
from argopy.plot import ArgoColors

ArgoColors().list_valid_known_colormaps
ArgoColors().known_colormaps.keys()

ArgoColors('data_mode')
ArgoColors('data_mode').cmap
ArgoColors('data_mode').definition

ArgoColors('Set2').cmap
ArgoColors('Spectral', N=25).cmap
```

Internals

- Because of the new `argopy.plot.ArgoColors`, the *argopy.plot.discrete_coloring* utility is deprecated in 0.1.13. Calling it will raise an error after argopy 0.1.14. (#245) by G. Maze
- New method to check status of web API: now allows for a keyword check rather than a simple url ping. This comes with 2 new utilities functions `utilities.urlhaskeyword()` and `utilities.isalive()`. (#247) by G. Maze.
- Removed dependency to Scikit-learn LabelEncoder (#239) by G. Maze

Breaking changes

- Data source `localftp` is deprecated and removed from **argopy**. It's been replaced by the `gdac` data source with

the appropriate `ftp` option. See *Data sources*. (#240) by G. Maze

Breaking changes with previous versions

- `argopy.utilities.ArgoNVSReferenceTables` methods `all_tbl` and `all_tbl_name` are now properties, not methods.

1.10.7 v0.1.12 (16 May 2022)

Internals

- Update `erddap` server from https://www.ifremer.fr/erddap to https://erddap.ifremer.fr/erddap. (@af5692f) by G. Maze

1.10.8 v0.1.11 (13 Apr. 2022)

Features and front-end API

- **New data source ``gdac`` to retrieve data from a GDAC compliant source**, for DataFetcher and IndexFetcher. You can specify the FTP source with the `ftp` fetcher option or with the argopy global option `ftp`. The FTP source support http, ftp or local files protocols. This fetcher is optimised if pyarrow is available, otherwise pandas dataframe are used. See update on *Data sources*. (#157) by G. Maze

```
from argopy import IndexFetcher
from argopy import DataFetcher
argo = IndexFetcher(src='gdac')
argo = DataFetcher(src='gdac')
argo = DataFetcher(src='gdac', ftp="https://data-argo.ifremer.fr")  # Default and␣
↪fastest !
argo = DataFetcher(src='gdac', ftp="ftp://ftp.ifremer.fr/ifremer/argo")
with argopy.set_options(src='gdac', ftp='ftp://usgodae.org/pub/outgoing/argo'):
    argo = DataFetcher()
```

Note: The new `gdac` fetcher uses Argo index to determine which profile files to load. Hence, this fetcher may show poor performances when used with a `region` access point. Don't hesitate to check *Performances* to try to improve performances, otherwise, we recommend to use a webAPI access point (`erddap` or `argovis`).

Warning: Since the new `gdac` fetcher can use a local copy of the GDAC ftp server, the legacy `localftp` fetcher is now deprecated. Using it will raise a error up to v0.1.12. It will then be removed in v0.1.13.

- **New dashboard for profiles and new 3rd party dashboards**. Calling on the data fetcher dashboard method will return the Euro-Argo profile page for a single profile. Very useful to look at the data before load. This comes with 2 new utilities functions to get Coriolis ID of profiles (`utilities.get_coriolis_profile_id()`) and to return the list of profile webpages (`utilities.get_ea_profile_page()`). (#198) by G. Maze.

```
from argopy import DataFetcher as ArgoDataFetcher
ArgoDataFetcher().profile(5904797, 11).dashboard()
```

```
from argopy.utilities import get_coriolis_profile_id, get_ea_profile_page
get_coriolis_profile_id([6902755, 6902756], [11, 12])
get_ea_profile_page([6902755, 6902756], [11, 12])
```

The new profile dashboard can also be accessed with:

```
import argopy
argopy.dashboard(5904797, 11)
```

We added the Ocean-OPS (former JCOMMOPS) dashboard for all floats and the Argo-BGC dashboard for BGC floats:

```
import argopy
argopy.dashboard(5904797, type='ocean-ops')
# or
argopy.dashboard(5904797, 12, type='bgc')
```

- **New utility :class:`argopy.utilities.ArgoNVSReferenceTables` to retrieve Argo Reference Tables**. (@cc8fdbe) by G. Maze.

```
from argopy.utilities import ArgoNVSReferenceTables
R = ArgoNVSReferenceTables()
R.all_tbl_name()
R.tbl(3)
R.tbl('R09')
```

Internals

- `gdac` and `localftp` data fetchers can return an index without loading the data. (#157) by G. Maze

```
from argopy import DataFetcher
argo = DataFetcher(src='gdac').float(6903076)
argo.index
```

- New index store design. A new index store is used by data and index `gdac` fetchers to handle access and search in Argo index csv files. It uses pyarrow table if available or pandas dataframe otherwise. More details at *Argo index store*. Directly using this index store is not recommended but provides better performances for expert users interested in Argo sampling analysis.

```
from argopy.stores.argo_index_pa import indexstore_pyarrow as indexstore
idx = indexstore(host="https://data-argo.ifremer.fr", index_file="ar_index_global_prof.
↪txt")  # Default
idx.load()
idx.search_lat_lon_tim([-60, -55, 40., 45., '2007-08-01', '2007-09-01'])
idx.N_MATCH  # Return number of search results
idx.to_dataframe()  # Convert search results to a dataframe
```

- Refactoring of CI tests to use more fixtures and pytest parametrize. (#157) by G. Maze
- Fix bug in erddap fata fetcher that was causing a *profile* request to do not account for cycle numbers. (@301e557) by G. Maze.

Breaking changes

- Index fetcher for local FTP no longer support the option `index_file`. The name of the file index is internally determined using the dataset requested: `ar_index_global_prof.txt` for `ds='phy'` and `argo_synthetic-profile_index.txt` for `ds='bgc'`. Using this option will raise a deprecation warning up to v0.1.12 and will then raise an error. (#157) by G. Maze
- Complete refactoring of the `argopy.plotters` module into `argopy.plot`. (#198) by G. Maze.
- Remove deprecation warnings for: 'plotters.plot_dac', 'plotters.plot_profilerType'. These now raise an error.

1.10.9 v0.1.10 (4 Mar. 2022)

Internals

- Update and clean up requirements. Remove upper bound on all dependencies (#182) by R. Abernathey.

1.10.10 v0.1.9 (19 Jan. 2022)

Features and front-end API

- **New method to preprocess data for OWC software**. This method can preprocessed Argo data and possibly create float_source/<WMO>.mat files to be used as inputs for OWC implementations in Matlab and Python. See the *Salinity calibration* documentation page for more. (#142) by G. Maze.

```
from argopy import DataFetcher as ArgoDataFetcher
ds = ArgoDataFetcher(mode='expert').float(6902766).load().data
ds.argo.create_float_source("float_source")
ds.argo.create_float_source("float_source", force='raw')
ds_source = ds.argo.create_float_source()
```

This new method comes with others methods and improvements:

- A new *Dataset.argo.filter_scalib_pres()* method to filter variables according to OWC salinity calibration software requirements,
- A new *Dataset.argo.groupby_pressure_bins()* method to subsample a dataset down to one value by pressure bins (a perfect alternative to interpolation on standard depth levels to precisely avoid interpolation…), see *Pressure levels: Group-by bins* for more help,
- An improved *Dataset.argo.filter_qc()* method to select which fields to consider (new option `QC_fields`),
- Add conductivity (`CNDC`) to the possible output of the `TEOS10` method.
- **New dataset properties** accessible from the *argo* xarray accessor: `N_POINTS`, `N_LEVELS`, `N_PROF`. Note that depending on the format of the dataset (a collection of points or of profiles) these values do or do not take into account NaN. These information are also visible by a simple print of the accessor. (#142) by G. Maze.

```
from argopy import DataFetcher as ArgoDataFetcher
ds = ArgoDataFetcher(mode='expert').float(6902766).load().data
ds.argo.N_POINTS
ds.argo.N_LEVELS
ds.argo.N_PROF
ds.argo
```

- **New plotter function** `argopy.plotters.open_sat_altim_report()` to insert the CLS Satellite Altimeter Report figure in a notebook cell. (#159) by G. Maze.

```
from argopy.plotters import open_sat_altim_report
open_sat_altim_report(6902766)
open_sat_altim_report([6902766, 6902772, 6902914])
open_sat_altim_report([6902766, 6902772, 6902914], embed='dropdown')  # Default
open_sat_altim_report([6902766, 6902772, 6902914], embed='slide')
open_sat_altim_report([6902766, 6902772, 6902914], embed='list')
open_sat_altim_report([6902766, 6902772, 6902914], embed=None)

from argopy import DataFetcher
```

(continues on next page)

(continued from previous page)

```
from argopy import IndexFetcher
DataFetcher().float([6902745, 6902746]).plot('qc_altimetry')
IndexFetcher().float([6902745, 6902746]).plot('qc_altimetry')
```

- **New utility method to retrieve topography**. The *argopy.TopoFetcher* will load the GEBCO topography for a given region. (#150) by G. Maze.

```
from argopy import TopoFetcher
box = [-75, -45, 20, 30]
ds = TopoFetcher(box).to_xarray()
ds = TopoFetcher(box, ds='gebco', stride=[10, 10], cache=True).to_xarray()
```

For convenience we also added a new property to the data fetcher that return the domain covered by the dataset.

```
loader = ArgoDataFetcher().float(2901623)
loader.domain  # Returns [89.093, 96.036, -0.278, 4.16, 15.0, 2026.0, numpy.datetime64(
↪'2010-05-14T03:35:00.000000000'),  numpy.datetime64('2013-01-01T01:45:00.000000000')]
```

- Update the documentation with a new section about *Data quality control*.

Internals

- Uses a new API endpoint for the `argovis` data source when fetching a `region`. More on this issue here. (#158) by G. Maze.
- Update documentation theme, and pages now use the xarray accessor sphinx extension. (#104) by G. Maze.
- Update Binder links to work without the deprecated Pangeo-Binder service. (#164) by G. Maze.

1.10.11 v0.1.8 (2 Nov. 2021)

Features and front-end API

- Improve plotting functions. All functions are now available for both the index and data fetchers. See the *Data visualisation* page for more details. Reduced plotting dependencies to Matplotlib only. **Argopy** will use Seaborn and/or Cartopy if available. (#56) by G. Maze.

```
from argopy import IndexFetcher as ArgoIndexFetcher
from argopy import DataFetcher as ArgoDataFetcher
obj = ArgoIndexFetcher().float([6902766, 6902772, 6902914, 6902746])
# OR
obj = ArgoDataFetcher().float([6902766, 6902772, 6902914, 6902746])

fig, ax = obj.plot()
fig, ax = obj.plot('trajectory')
fig, ax = obj.plot('trajectory', style='white', palette='Set1', figsize=(10,6))
fig, ax = obj.plot('dac')
fig, ax = obj.plot('institution')
fig, ax = obj.plot('profiler')
```

- New methods and properties for data and index fetchers. (#56) by G. Maze. The *argopy.DataFetcher.load()* and *argopy.IndexFetcher.load()* methods internally call on the *to_xarray()* methods and store results in the fetcher instance. The *argopy.DataFetcher.to_xarray()* will trigger a fetch on every call, while the *argopy.DataFetcher.load()* will not.

```
from argopy import DataFetcher as ArgoDataFetcher
loader = ArgoDataFetcher().float([6902766, 6902772, 6902914, 6902746])
loader.load()
loader.data
loader.index
loader.to_index()
```

```
from argopy import IndexFetcher as ArgoIndexFetcher
indexer = ArgoIndexFetcher().float([6902766, 6902772])
indexer.load()
indexer.index
```

- Add optional speed of sound computation to xarray accessor teos10 method. (#90) by G. Maze.
- Code spell fixes (#89) by K. Schwehr.

Internals

- Check validity of access points options (WMO and box) in the facade, no checks at the fetcher level. (#92) by G. Maze.
- More general options. Fix #91. (#102) by G. Maze.
  - `trust_env` to allow for local environment variables to be used by fsspec to connect to the internet. Useful for those using a proxy.
- Documentation on *Read The Docs* now uses a pip environment and get rid of memory eager conda. (#103) by G. Maze.
- `xarray.Dataset` argopy accessor `argo` has a clean documentation.

Breaking changes with previous versions

- Drop support for python 3.6 and older. Lock range of dependencies version support.
- In the plotters module, the `plot_dac` and `plot_profilerType` functions have been replaced by `bar_plot`. (#56) by G. Maze.

Internals

- Internal logging available and upgrade dependencies version support (#56) by G. Maze. To see internal logs, you can set-up your application like this:

```
import logging
DEBUGFORMATTER = '%(asctime)s [%(levelname)s] [%(name)s] %(filename)s:%(lineno)d:
↪%(message)s'
logging.basicConfig(
    level=logging.DEBUG,
    format=DEBUGFORMATTER,
    datefmt='%m/%d/%Y %I:%M:%S %p',
    handlers=[logging.FileHandler("argopy.log", mode='w')]
)
```

1.10.12 v0.1.7 (4 Jan. 2021)

Long due release !

Features and front-end API

- Live monitor for the status (availability) of data sources. See documentation page on *Status of sources*. (#36) by G. Maze.

```
import argopy
argopy.status()
# or
argopy.status(refresh=15)
```

- Optimise large data fetching with parallelization, for all data fetchers (erddap, localftp and argovis). See documentation page on *Parallel data fetching*. Two parallel methods are available: multi-threading or multi-processing. (#28) by G. Maze.

```
from argopy import DataFetcher as ArgoDataFetcher
loader = ArgoDataFetcher(parallel=True)
loader.float([6902766, 6902772, 6902914, 6902746]).to_xarray()
loader.region([-85,-45,10.,20.,0,1000.,'2012-01','2012-02']).to_xarray()
```

Breaking changes with previous versions

- In the teos10 xarray accessor, the `standard_name` attribute will now be populated using values from the CF Standard Name table if one exists. The previous values of `standard_name` have been moved to the `long_name` attribute. (#74) by A. Barna.
- The unique resource identifier property is now named `uri` for all data fetchers, it is always a list of strings.

Internals

- New `open_mfdataset` and `open_mfjson` methods in Argo stores. These can be used to open, pre-process and concatenate a collection of paths both in sequential or parallel order. (#28) by G. Maze.
- Unit testing is now done on a controlled conda environment. This allows to more easily identify errors coming from development vs errors due to dependencies update. (#65) by G. Maze.

1.10.13 v0.1.6 (31 Aug. 2020)

- **JOSS paper published**. You can now cite argopy with a clean reference. (#30) by G. Maze and K. Balem.

Maze G. and Balem K. (2020). argopy: A Python library for Argo ocean data analysis. *Journal of Open Source Software*, 5(52), 2425 doi: 10.21105/joss.02425.

1.10.14 v0.1.5 (10 July 2020)

Features and front-end API

- A new data source with the **argovis** data fetcher, all access points available (#24). By T. Tucker and G. Maze.

```
from argopy import DataFetcher as ArgoDataFetcher
loader = ArgoDataFetcher(src='argovis')
loader.float(6902746).to_xarray()
loader.profile(6902746, 12).to_xarray()
loader.region([-85,-45,10.,20.,0,1000.,'2012-01','2012-02']).to_xarray()
```

- Easily compute TEOS-10 variables with new argo accessor function **teos10**. This needs gsw to be installed. (#37) By G. Maze.

```
from argopy import DataFetcher as ArgoDataFetcher
ds = ArgoDataFetcher().region([-85,-45,10.,20.,0,1000.,'2012-01','2012-02']).to_xarray()
ds = ds.argo.teos10()
ds = ds.argo.teos10(['PV'])
ds_teos10 = ds.argo.teos10(['SA', 'CT'], inplace=False)
```

- **argopy** can now be installed with conda (#29, #31, #32). By F. Fernandes.

```
conda install -c conda-forge argopy
```

Breaking changes with previous versions

- The `local_ftp` option of the `localftp` data source must now points to the folder where the `dac` directory is found. This breaks compatibility with rsynced local FTP copy because rsync does not give a `dac` folder (e.g. #33). An instructive error message is raised to notify users if any of the DAC name is found at the n-1 path level. (#34).

Internals

- Implement a webAPI availability check in unit testing. This allows for more robust `erddap` and `argovis` tests that are not only based on internet connectivity only. (@5a46a39).

1.10.15 v0.1.4 (24 June 2020)

Features and front-end API

- Standard levels interpolation method available in **standard** user mode (#23). By K. Balem.

```
ds = ArgoDataFetcher().region([-85,-45,10.,20.,0,1000.,'2012-01','2012-12']).to_xarray()
ds = ds.argo.point2profile()
ds_interp = ds.argo.interp_std_levels(np.arange(0,900,50))
```

- Insert in a Jupyter notebook cell the Euro-Argo fleet monitoring dashboard page, possibly for a specific float (#20). By G. Maze.

```
import argopy
argopy.dashboard()
# or
argopy.dashboard(wmo=6902746)
```

- The `localftp` index and data fetcher now have the `region` and `profile` access points available (#25). By G. Maze.

Breaking changes with previous versions

[None]

Internals

- Now uses fsspec as file system for caching as well as accessing local and remote files (#19). This closes issues #12, #15 and #17. **argopy** fetchers must now use (or implement if necessary) one of the internal file systems available in the new module `argopy.stores`. By G. Maze.
- Erddap fetcher now uses netcdf format to retrieve data (#19).

1.10.16 v0.1.3 (15 May 2020)

Features and front-end API

- New `index` fetcher to explore and work with meta-data (#6). By K. Balem.

```
from argopy import IndexFetcher as ArgoIndexFetcher
idx = ArgoIndexFetcher().float(6902746)
idx.to_dataframe()
idx.plot('trajectory')
```

The `index` fetcher can manage caching and works with both Erddap and localftp data sources. It is basically the same as the data fetcher, but do not load measurements, only meta-data. This can be very useful when looking for regional sampling or trajectories.

Tip: Performance: we recommend to use the `localftp` data source when working this `index` fetcher because the `erddap` data source currently suffers from poor performances. This is linked to #16 and is being addressed by Ifremer.

The `index` fetcher comes with basic plotting functionalities with the *argopy.IndexFetcher.plot()* method to rapidly visualise measurement distributions by DAC, latitude/longitude and floats type.

Warning: The design of plotting and visualisation features in `argopy` is constantly evolving, so this may change in future releases.

- Real documentation written and published (#13). By G. Maze.
- The *argopy.DataFetcher* now has a *argopy.DataFetcher.to_dataframe()* method to return a `pandas.DataFrame`.
- Started a draft for JOSS (@1e37df4).
- New utilities function: `argopy.utilities.open_etopo1()`, *argopy.show_versions()*.

Breaking changes with previous versions

- The `backend` option in data fetchers and the global option `datasrc` have been renamed to `src`. This makes the code more coherent (@ec6b32e).

Code management

- Add Pypi automatic release publishing with github actions (@c430788)
- Remove Travis CI, fully adopt Github actions (@c455742)
- Improved unit testing (@e9555d1, @4b60ede, @34abf49)

1.10.17 v0.1.2 (15 May 2020)

We didn't like this one this morning, so we move one to the next one !

1.10.18 v0.1.1 (3 Apr. 2020)

Features and front-end API

- Added new data fetcher backend `localftp` in DataFetcher (@c5f7cb6):

```
from argopy import DataFetcher as ArgoDataFetcher
argo_loader = ArgoDataFetcher(backend='localftp', path_ftp='/data/Argo/ftp_copy')
argo_loader.float(6902746).to_xarray()
```

- Introduced global `OPTIONS` to set values for: cache folder, dataset (eg:*phy* or *bgc*), local ftp path, data fetcher (*erddap* or *localftp*) and user level (*standard* or *expert*). Can be used in context *with* (@83ccfb5):

```
with argopy.set_options(mode='expert', datasrc='erddap'):
    ds = argopy.DataFetcher().float(3901530).to_xarray()
```

- Added a `argopy.tutorial` module to be able to load sample data for documentation and unit testing (@4af09b5):

```
ftproot, flist = argopy.tutorial.open_dataset('localftp')
txtfile = argopy.tutorial.open_dataset('weekly_index_prof')
```

- Improved xarray *argo* accessor. Added methods for casting data types, to filter variables according to data mode, to filter variables according to quality flags. Useful methods to transform collection of points into collection of profiles, and vice versa (@14cda55):

```
ds = argopy.DataFetcher().float(3901530).to_xarray() # get a collection of points
dsprof = ds.argo.point2profile() # transform to profiles
ds = dsprof.argo.profile2point() # transform to points
```

- Changed License from MIT to Apache (@25f90c9)

Internal machinery

- Add `__all__` to control `from argopy import *` (@83ccfb5)
- All data fetchers inherit from class `ArgoDataFetcherProto` in `proto.py` (@44f45a5)
- Data fetchers use default options from global OPTIONS
- In Erddap fetcher: methods to cast data type, to filter by data mode and by QC flags are now delegated to the xarray argo accessor methods.
- Data fetchers methods to filter variables according to user mode are using variable lists defined in utilities.
- `argopy.utilities` augmented with listing functions of: backends, standard variables and multiprofile files variables.
- Introduce custom errors in errors.py (@2563c9f)
- Front-end API ArgoDataFetcher uses a more general way of auto-discovering fetcher backend and their access points. Turned of the `deployments` access point, waiting for the index fetcher to do that.
- Improved xarray *argo* accessor. More reliable `point2profile` and data type casting with `cast_type`

Code management

- Add CI with github actions (@ecbf9ba)
- Contribution guideline for data fetchers (@b332495)
- Improve unit testing (all along commits)
- Introduce code coverage (@b490ab5)
- Added explicit support for python 3.6 , 3.7 and 3.8 (@58f60fe)

1.10.19 v0.1.0 (17 Mar. 2020)

- Initial release.
- Erddap data fetcher

1.11 Contributing to argopy

Table of contents:

First off, thanks for taking the time to contribute!

Note: Large parts of this document came from the Xarray and Pandas contributing guides.

If you seek **support** for your argopy usage or if you don't want to read this whole thing and just have a question: visit our Discussion forum.

1.11.1 Where to start?

All contributions, bug reports, bug fixes, documentation improvements, enhancements, and ideas are welcome.

If you are brand new to *argopy* or open source development, we recommend going through the GitHub "issues" tab to find issues that interest you. There are a number of issues listed under Documentation and Good first issues where you could start out. Once you've found an interesting issue, you can return here to get your development environment setup.

Please don't file an issue to ask a question, instead visit our Discussion forum. where a number of items are listed under Documentation and Good first issue

1.11.2 Bug reports and enhancement requests

Bug reports are an important part of making *argopy* more stable. Having a complete bug report will allow others to reproduce the bug and provide insight into fixing. See this stackoverflow article for tips on writing a good bug report.

Trying the bug producing code out on the *master* branch is often a worthwhile exercise to confirm the bug still exists. It is also worth searching existing bug reports and pull requests to see if the issue has already been reported and/or fixed.

Bug reports must:

1. Include a short, self contained Python snippet reproducing the problem. You can format the code nicely by using GitHub Flavored Markdown:

```
```python
>>> import argopy as ar
>>> ds = ar.DataFetcher(backend='erddap').float(5903248).to_xarray()
...
```
```

2. Include the full version string of *argopy* and its dependencies. You can use the built in function:

```
>>> import argopy
>>> argopy.show_versions()
```

3. Explain why the current behavior is wrong/not desired and what you expect instead.

The issue will then show up to the `argopy` community and be open to comments/ideas from others.

Click here to open an issue with the specific bug reporting template

1.11.3 Contributing to the documentation

If you're not the developer type, contributing to the documentation is still of huge value. You don't even have to be an expert on *argopy* to do so! In fact, there are sections of the docs that are worse off after being written by experts. If something in the docs doesn't make sense to you, updating the relevant section after you figure it out is a great way to ensure it will help the next person.

Documentation:

About the *argopy* documentation

The documentation is written in **reStructuredText**, which is almost like writing in plain English, and built using Sphinx. The Sphinx Documentation has an excellent introduction to reST. Review the Sphinx docs to perform more complex changes to the documentation as well.

Some other important things to know about the docs:

- The *argopy* documentation consists of two parts: the docstrings in the code itself and the docs in this folder `argopy/docs/`.

  The docstrings are meant to provide a clear explanation of the usage of the individual functions, while the documentation in this folder consists of tutorial-like overviews per topic together with some other information (what's new, installation, etc).

- The docstrings follow the **Numpy Docstring Standard**, which is used widely in the Scientific Python community. This standard specifies the format of the different sections of the docstring. See this document for a detailed explanation, or look at some of the existing functions to extend it in a similar manner.

- The tutorials make use of the ipython directive sphinx extension. This directive lets you put code in the documentation which will be run during the doc build. For example:

  ```
  .. ipython:: python

      x = 2
      x ** 3
  ```

  will be rendered as:

  ```
  In [1]: x = 2

  In [2]: x ** 3
  Out[2]: 8
  ```

  Almost all code examples in the docs are run (and the output saved) during the doc build. This approach means that code examples will always be up to date, but it does make the doc building a bit more complex.

- Our API documentation in `docs/api.rst` houses the auto-generated documentation from the docstrings. For classes, there are a few subtleties around controlling which methods and attributes have pages auto-generated.

Every method should be included in a `toctree` in `api.rst`, else Sphinx will emit a warning.

How to build the *argopy* documentation

Requirements

Make sure to follow the instructions on *creating a development environment below* and use the specific environment `argopy-docs`:

```
$ ./ci/envs_manager -i argopy-docs
$ conda activate argopy-docs
$ pip install -e .
$ pip install -r docs/requirements.txt
```

Building the documentation

Navigate to your local `argopy/docs/` directory in the console and run:

```
make html
```

Then you can find the HTML output in the folder `argopy/docs/_build/html/`.

The first time you build the docs, it will take quite a while because it has to run all the code examples and build all the generated docstring pages. In subsequent evocations, sphinx will try to only build the pages that have been modified.

If you want to do a full clean build, do:

```
make clean
make html
```

1.11.4 Working with the code

Development workflow

Anyone interested in helping to develop argopy needs to create their own fork of our *git repository*. (Follow the github forking instructions. You will need a github account.)

Clone your fork on your local machine.

```
$ git clone git@github.com:USERNAME/argopy
```

(In the above, replace USERNAME with your github user name.)

Then set your fork to track the upstream argopy repo.

```
$ cd argopy
$ git remote add upstream git://github.com/euroargodev/argopy.git
```

You will want to periodically sync your master branch with the upstream master.

```
$ git fetch upstream
$ git rebase upstream/master
```

Never make any commits on your local master branch. Instead open a feature branch for every new development task.

```
$ git checkout -b cool_new_feature
```

(Replace *cool_new_feature* with an appropriate description of your feature.) At this point you work on your new feature, using *git add* to add your changes. When your feature is complete and well tested, commit your changes

```
$ git commit -m 'did a bunch of great work'
```

and push your branch to github.

```
$ git push origin cool_new_feature
```

At this point, you go find your fork on github.com and create a pull request. Clearly describe what you have done in the comments. If your pull request fixes an issue or adds a useful new feature, the team will gladly merge it.

After your pull request is merged, you can switch back to the master branch, rebase, and delete your feature branch. You will find your new feature incorporated into argopy.

```
$ git checkout master
$ git fetch upstream
$ git rebase upstream/master
$ git branch -d cool_new_feature
```

Virtual environment

We created a short command line script to help manage argopy virtual environments. It's available in the "ci" folder of the repository.

```
$ ./ci/envs_manager -h

Manage argopy related Conda environments

Syntax: manage_ci_envs [-hl] [-d] [-rik]
options:
h     Print this Help
l     List all available environments
d     Dry run, just list what the script would do

r     Remove an environment
i     Install an environment (start by removing it if it's already installed)
k     Install an environment as a Jupyter kernel

$ ./ci/envs_manager -l

Available environments:
 argopy-docs
 argopy-py38-core-free
 argopy-py37-all-free
 argopy-py38-all-pinned
 argopy-py37-core-free
 argopy-py39-all-free
```

(continues on next page)

(continued from previous page)

```
argopy-py39-core-free
argopy-py39-all-pinned
argopy-py38-all-free
argopy-py38-core-pinned
```

Then, you can simply install the default dev environment like this:

> $ ./ci/envs_manager -i argopy-py38-all-pinned $ conda activate argopy-py38-all-pinned $ pip install -e . $ python -c 'import argopy; argopy.show_versions()'

Code standards

Writing good code is not just about what you write. It is also about *how* you write it. During Continuous Integration testing, several tools will be run to check your code for stylistic errors. Generating any warnings will cause the test to fail. Thus, good style is a requirement for submitting code to *argopy*.

Code Formatting

argopy uses several tools to ensure a consistent code format throughout the project:

- Flake8 for general code quality

`pip`:

```
pip install flake8
```

and then run from the root of the argopy repository:

```
flake8
```

to qualify your code.

1.11.5 Contributing to the code base

Code Base:

- *Data fetchers*
  - *Introduction*
  - *Detailed guideline*
    - *Inheritance*
    - *Auto-discovery of fetcher properties*
    - *Auto-discovery of fetcher access points*
    - *Internal File systems*
    - *Output data format*

Data fetchers

Introduction

If you want to add your own data fetcher for a new service, then, keep in mind that:

- Data fetchers are responsible for:
  - loading all available data from a given source and providing at least a `to_xarray()` method
  - making data compliant to Argo standards (data type, variable name, attributes, etc ...)
- Data fetchers must:
  - inherit from the `argopy.data_fetchers.proto.ArgoDataFetcherProto`
  - provide parameters:
    - `access_points`, eg: ['wmo', 'box']
    - `exit_formats`, eg: ['xarray']
    - `dataset_ids`, eg: ['phy', 'ref', 'bgc']
  - provides the facade API (`argopy.fetchers.ArgoDataFetcher`) methods to filter data according to user level or requests. These must includes:
    - `filter_data_mode()`
    - `filter_qc()`
    - `filter_variables()`

It is the responsibility of the facade API (`argopy.fetchers.ArgoDataFetcher`) to run filters according to user level or requests, not the data fetcher.

Detailed guideline

A new data fetcher must comply with:

Inheritance

Inherit from the `argopy.data_fetchers.proto.ArgoDataFetcherProto`. This enforces minimal internal design compliance.

Auto-discovery of fetcher properties

The new fetcher must come with the `access_points`, `exit_formats` and `dataset_ids` properties at the top of the file, e.g.:

```
access_points = ['wmo' ,'box']
exit_formats = ['xarray']
dataset_ids = ['phy', 'bgc']  # First is default
```

Values depend on what the new access point can return and what you want to implement. A good start is with the `wmo` access point and the `phy` dataset ID. The `xarray` data format is the minimum required. These variables are used by the facade to auto-discover the fetcher capabilities. The `dataset_ids` property is used to determine which variables can be retrieved.

Auto-discovery of fetcher access points

The new fetcher must come at least with a `Fetch_box` or `Fetch_wmo` class, basically one for each of the `access_points` listed as properties. More generally we may have a main class that provides the key functionality to retrieve data from the source, and then classes for each of the `access_points` of your fetcher. This pattern could look like this:

```
class NewDataFetcher(ArgoDataFetcherProto)
class Fetch_wmo(NewDataFetcher)
class Fetch_box(NewDataFetcher)
```

It could also be like:

```
class Fetch_wmo(ArgoDataFetcherProto)
class Fetch_box(ArgoDataFetcherProto)
```

Note that the class names `Fetch_wmo` and `Fetch_box` must not change, this is also used by the facade to auto-discover the fetcher capabilities.

Fetch_wmo is used to retrieve platforms and eventually profiles data. It must take in the `__init__()` method a `WMO` and a `CYC` as first and second options. `WMO` is always passed, `CYC` is optional. These are passed by the facade to implement the `fetcher.float` and `fetcher.profile` methods. When a float is requested, the `CYC` option is not passed by the facade. Last, `WMO` and `CYC` are either a single integer or a list of integers: this means that `Fetch_wmo` must be able to handle more than one float/platform retrieval.

Fetch_box is used to retrieve a rectangular domain in space and time. It must take in the `__init__()` method a `BOX` as first option that is passed a list(lon_min: float, lon_max: float, lat_min: float, lat_max: float, pres_min: float, pres_max: float, date_min: str, date_max: str) from the facade. The two bounding dates [date_min and date_max] should be optional (if not specified, the entire time series is requested by the user).

Internal File systems

All http requests must go through the internal `httpstore`, an internal wrapper around fsspec that allows to manage request caching very easily. You can simply use it this way for json requests:

```
from argopy.stores import httpstore
with httpstore(timeout=120).open("https://argovis.colorado.edu/catalog/profiles/5904797_
↪12") as of:
   profile = json.load(of)
```

Output data format

Last but not least, about the output data. In **argopy**, we want to provide data for both expert and standard users. This is explained and illustrated in the documentation here. This means for a new data fetcher that the data content should be curated and clean of any internal/jargon variables that is not part of the Argo ADMT vocabulary. For instance, variables like: `bgcMeasKeys` or `geoLocation` are not allowed. This will ensure that whatever the data source set by users, the output xarray or dataframe will be formatted and contain the same variables. This will also ensure that other argopy features can be used on the new fetcher output, like plotting or xarray data manipulation.

1.12 API reference

This page provides an auto-generated summary of argopy's API. For more details and examples, refer to the relevant chapters in the main part of the documentation.

1.12.1 Argo Data Fetchers

DataFetcher	alias of `ArgoDataFetcher`
IndexFetcher	alias of `ArgoIndexFetcher`

argopy.DataFetcher

DataFetcher

alias of ArgoDataFetcher

argopy.IndexFetcher

IndexFetcher

alias of ArgoIndexFetcher

Data selection methods

DataFetcher.region()	Space/time domain data fetcher
DataFetcher.float()	Float data fetcher
DataFetcher.profile()	Profile data fetcher

argopy.DataFetcher.region

DataFetcher.**region**()

Space/time domain data fetcher

Parameters

box (*list()*) –

Define the domain to load Argo data for. The box list is made of:

- lon_min: float, lon_max: float,
- lat_min: float, lat_max: float,
- dpt_min: float, dpt_max: float,
- date_min: str (optional), date_max: str (optional)

Longitude, latitude and pressure bounds are required, while the two bounding dates are optional. If bounding dates are not specified, the entire time series is fetched. Eg: [-60, -55, 40., 45., 0., 10., '2007-08-01', '2007-09-01']

Returns

A data source fetcher for a space/time domain

Return type

argopy.fetchers.ArgoDataFetcher

argopy.DataFetcher.float

DataFetcher.**float**()

Float data fetcher

Parameters
wmo (*int, list(int)*) – Define the list of Argo floats to load data for. This is a list of integers with WMO float identifiers. WMO is the World Meteorological Organization.

Returns
A data source fetcher for all float profiles

Return type
argopy.fetchers.ArgoDataFetcher.float

argopy.DataFetcher.profile

DataFetcher.**profile**()

Profile data fetcher

Parameters

- **wmo** (*int, list(int)*) – Define the list of Argo floats to load data for. This is a list of integers with WMO float identifiers. WMO is the World Meteorological Organization.
- **cyc** (*list(int)*) – Define the list of cycle numbers to load for each Argo floats listed in wmo.

Returns
A data source fetcher for specific float profiles

Return type
argopy.fetchers.ArgoDataFetcher.profile

IndexFetcher.region()	Space/time domain index fetcher
IndexFetcher.float()	Float index fetcher
IndexFetcher.profile()	Profile index fetcher

argopy.IndexFetcher.region

IndexFetcher.**region**()

Space/time domain index fetcher

Parameters
box (*list()*) –

Define the domain to load Argo index for. The box list is made of:

- lon_min: float, lon_max: float,
- lat_min: float, lat_max: float,
- date_min: str (optional), date_max: str (optional)

Longitude and latitude bounds are required, while the two bounding dates are optional. If bounding dates are not specified, the entire time series is fetched. Eg: [-60, -55, 40., 45., '2007-08-01', '2007-09-01']

Returns
An index fetcher initialised for a space/time domain

Return type
`argopy.fetchers.ArgoIndexFetcher`

> **Warning:** Note that the box option for an index fetcher does not have pressure bounds, contrary to the data fetcher.

argopy.IndexFetcher.float

`IndexFetcher.float()`
Float index fetcher

Parameters
wmo (*list(int)*) – Define the list of Argo floats to load data for. This is a list of integers with WMO numbers.

Returns
An index fetcher initialised for specific floats

Return type
`argopy.fetchers.ArgoIndexFetcher`

argopy.IndexFetcher.profile

`IndexFetcher.profile()`
Profile index fetcher

Parameters

- **wmo** (*int, list(int)*) – Define the list of Argo floats to load index for. This is a list of integers with WMO float identifiers. WMO is the World Meteorological Organization.
- **cyc** (*list(int)*) – Define the list of cycle numbers to load for each Argo floats listed in `wmo`.

Returns
An index fetcher initialised for specific float profiles

Return type
`argopy.fetchers.ArgoIndexFetcher`

Data access methods

DataFetcher.load([force])	Fetch data (and compute a profile index) if not already in memory
DataFetcher.to_xarray(**kwargs)	Fetch and return data as xarray.DataSet
DataFetcher.to_dataframe(**kwargs)	Fetch and return data as pandas.Dataframe
DataFetcher.to_index([full, coriolis_id])	Create a profile index of Argo data, fetch data if necessary

argopy.DataFetcher.load

DataFetcher.**load**(*force: bool = False*, ***kwargs*)

Fetch data (and compute a profile index) if not already in memory

Apply the default to_xarray() and to_index() methods and store results in memory. You can access loaded measurements structure with the *data* and *index* properties.

Parameters
force (*bool*) – Force fetching data even if not already in memory, default is False.

Returns
Data fetcher with *data* and *index* properties in memory

Return type
argopy.fetchers.ArgoDataFetcher

Examples

```
>>> ds = ArgoDataFetcher().profile(6902746, 34).load().data
>>> df = ArgoDataFetcher().float(6902746).load().index
```

argopy.DataFetcher.to_xarray

DataFetcher.**to_xarray**(***kwargs*)

Fetch and return data as xarray.DataSet

Trigger a fetch of data by the specified source and access point.

Returns
Fetched data

Return type
xarray.DataSet

argopy.DataFetcher.to_dataframe

DataFetcher.**to_dataframe**(***kwargs*)

Fetch and return data as pandas.Dataframe

Trigger a fetch of data by the specified source and access point.

Returns
Fetched data

Return type
pandas.DataFrame

argopy.DataFetcher.to_index

DataFetcher.**to_index**(*full: bool = False*, *coriolis_id: bool = False*)

Create a profile index of Argo data, fetch data if necessary

Build an Argo-like index of profiles from fetched data.

Parameters

- **full** (*bool, default: False*) – If possible, should extract a reduced index (only space/time/wmo/cyc) from fetched profiles, otherwise a full index, as returned by an IndexFetcher.
- **coriolis_id** (*bool, default: False*) – Add a column to the index with the Coriolis ID of profiles

Returns

Argo-like index of fetched data

Return type

pandas.DataFrame

IndexFetcher.load([force])	Load index in memory
IndexFetcher.to_xarray(**kwargs)	Fetch and return index data as xarray DataSet
IndexFetcher.to_dataframe(**kwargs)	Fetch and return index data as pandas Dataframe
IndexFetcher.to_csv([file])	Fetch and save index data as csv in a file

argopy.IndexFetcher.load

IndexFetcher.**load**(*force: bool = False*)

Load index in memory

Apply the default to_dataframe() method and store results in memory. You can access the index array with the *index* property:

```
>>> df = ArgoIndexFetcher().float(6902746).load().index
```

Parameters

force (*bool*) – Force loading, default is False.

Returns

Index fetcher with *index* property in memory

Return type

argopy.fetchers.ArgoIndexFetcher.float

argopy.IndexFetcher.to_xarray

IndexFetcher.**to_xarray**(***kwargs*)

Fetch and return index data as xarray DataSet

This is a shortcut to .load().index.to_xarray()

Return type
xarray.DataSet

argopy.IndexFetcher.to_dataframe

IndexFetcher.**to_dataframe**(***kwargs*)

Fetch and return index data as pandas Dataframe

Return type
pandas.DataFrame

argopy.IndexFetcher.to_csv

IndexFetcher.**to_csv**(*file: str = 'output_file.csv'*)

Fetch and save index data as csv in a file

Notes

```
>>> idx.to_csv()
is a shortcut to:
>>> idx.load().index.to_csv()
```

Since the `index` property is a `pandas.DataFrame`, this is currently a short cut to `pandas.DataFrame.to_index()`

Return type
None

Data visualisation methods

DataFetcher.plot([ptype])	Create custom plots from this fetcher data or index.
DataFetcher.dashboard(**kw)	Open access point dashboard.
IndexFetcher.plot([ptype])	Create custom plots from this fetcher index.

argopy.DataFetcher.plot

DataFetcher.**plot**(*ptype: str = 'trajectory'*, ***kwargs*)

Create custom plots from this fetcher data or index.

This is basically shortcuts to some plotting submodules:

- **trajectory** calls `argopy.plot.plot_trajectory` with index DataFrame
- **profiler** or **dac** calls *argopy.plot.bar_plot* with index DataFrame
- **qc_altimetry** calls *argopy.plot.open_sat_altim_report* with data unique list of `PLATFORM_NUMBER`

Parameters

- **ptype** (*str, default: 'trajectory'*) – Plot type, one of the following: `trajectory`, `profiler`, `dac` or `qc_altimetry`.
- **kwargs** – Other arguments passed to the plotting submodule.

Returns

- **fig** (`matplotlib.figure.Figure`)
- **ax** (`matplotlib.axes.Axes`)

Warning: Calling this method will automatically trigger a call to the *argopy.DataFetcher.load* method.

argopy.DataFetcher.dashboard

DataFetcher.**dashboard**(***kw*)

Open access point dashboard.

See also:

argopy.dashboard

argopy.IndexFetcher.plot

IndexFetcher.**plot**(*ptype: str = 'trajectory'*, ***kwargs*)

Create custom plots from this fetcher index.

This is basically shortcuts to some plotting submodules:

- **trajectory** calls `argopy.plot.plot_trajectory` with index DataFrame
- **profiler** or **dac** calls *argopy.plot.bar_plot* with index DataFrame
- **qc_altimetry** calls *argopy.plot.open_sat_altim_report* with index unique list of `wmo`

Parameters

- **ptype** (*str, default: 'trajectory'*) – Plot type, one of the following: `trajectory`, `profiler`, `dac` or `qc_altimetry`.
- **kwargs** – Other arguments passed to the plotting submodule.

Returns

- **fig** (matplotlib.figure.Figure)
- **ax** (matplotlib.axes.Axes)

Warning: Calling this method will automatically trigger a call to the *argopy.IndexFetcher.load* method.

Properties

DataFetcher.data	Data structure
DataFetcher.index	Index structure, as returned by the to_index method
DataFetcher.domain	Space/time domain of the dataset
DataFetcher.uri	List of resources to load for a request
IndexFetcher.index	Index structure

argopy.DataFetcher.data

property DataFetcher.**data**

Data structure

Returns
Fetched data

Return type
xarray.DataArray

argopy.DataFetcher.index

property DataFetcher.**index**

Index structure, as returned by the to_index method

Returns
Argo-like index of fetched data

Return type
pandas.DataFrame

argopy.DataFetcher.domain

property DataFetcher.**domain**

Space/time domain of the dataset

This is different from a usual `box` because dates are in numpy.datetime64 format.

argopy.DataFetcher.uri

property DataFetcher.**uri**

List of resources to load for a request

This can be a list of paths or urls, depending on the data source selected.

Returns
List of resources used to fetch data

Return type
list(str)

argopy.IndexFetcher.index

property IndexFetcher.**index**

Index structure

Returns
Argo-like index of fetched data

Return type
pandas.DataFrame

1.12.2 Utilities for Argo related data

status	alias of monitor_status
ArgoIndex([host, index_file, convention, ...])	Argo GDAC index store
ArgoNVSReferenceTables([nvs, cache, cachedir])	Argo Reference Tables
OceanOPSDeployments([box, deployed_only])	Use the OceanOPS API for metadata access to retrieve Argo floats deployment information.
CTDRefDataFetcher	alias of Fetch_box
TopoFetcher(box[, ds, cache, cachedir, ...])	Fetch topographic data through an ERDDAP server for an ocean rectangle
ArgoDocs([docid, cache])	ADMT documentation helper class
related.get_coriolis_profile_id(WMO[, CYC])	Return a pandas.DataFrame with CORIOLIS ID of WMO/CYC profile pairs
related.get_ea_profile_page(WMO[, CYC])	Return a list of URL

argopy.status

status

alias of monitor_status

argopy.ArgoIndex

class ArgoIndex(*host: str = 'https://data-argo.ifremer.fr'*, *index_file: str = 'ar_index_global_prof.txt'*, *convention: str | None = None*, *cache: bool = False*, *cachedir: str = ''*, *timeout: int = 0*)

Argo GDAC index store

If Pyarrow is available, this class will use `pyarrow.Table` as internal storage format; otherwise, a `pandas.DataFrame` will be used.

You can use the exact index file names or keywords:

- `core` for the `ar_index_global_prof.txt` index file,
- `bgc-b` for the `argo_bio-profile_index.txt` index file,
- `bgc-s` for the `argo_synthetic-profile_index.txt` index file.

Examples

An index store is instantiated with a host (any access path, local, http or ftp) and an index file:

```
>>> idx = ArgoIndex()
>>> idx = ArgoIndex(host="https://data-argo.ifremer.fr")  # Default host
>>> idx = ArgoIndex(host="ftp://ftp.ifremer.fr/ifremer/argo", index_file="ar_index_
↪global_prof.txt")  # Default index
>>> idx = ArgoIndex(index_file="bgc-s")  # Use keywords instead of exact file names
>>> idx = ArgoIndex(host="https://data-argo.ifremer.fr", index_file="bgc-b",␣
↪cache=True)  # Use cache for performances
>>> idx = ArgoIndex(host=".", index_file="dummy_index.txt", convention="core")  #␣
↪Load your own index
```

Full index methods and properties:

```
>>> idx.load()
>>> idx.load(nrows=12)  # Only load the first N rows of the index
>>> idx.to_dataframe(index=True)  # Convert index to user-friendly :class:`pandas.
↪DataFrame`
>>> idx.to_dataframe(index=True, nrows=2)  # Only returns the first nrows of the␣
↪index
>>> idx.N_RECORDS  # Shortcut for length of 1st dimension of the index array
>>> idx.index  # internal storage structure of the full index (:class:`pyarrow.
↪Table` or :class:`pandas.DataFrame`)
>>> idx.shape  # shape of the full index array
>>> idx.uri_full_index  # List of absolute path to files from the full index table␣
↪column 'file'
```

Search methods:

```
>>> idx.search_wmo(1901393)
>>> idx.search_cyc(1)
>>> idx.search_wmo_cyc(1901393, [1,12])
>>> idx.search_tim([-60, -55, 40., 45., '2007-08-01', '2007-09-01'])  # Take an␣
↪index BOX definition
>>> idx.search_lat_lon([-60, -55, 40., 45., '2007-08-01', '2007-09-01'])  # Take an␣
↪index BOX definition
```

(continues on next page)

(continued from previous page)

```
>>> idx.search_lat_lon_tim([-60, -55, 40., 45., '2007-08-01', '2007-09-01'])  #␣
↪Take an index BOX definition
>>> idx.search_params(['C1PHASE_DOXY', 'DOWNWELLING_PAR'])  # Take a list of␣
↪strings, only for BGC index !
>>> idx.search_parameter_data_mode({'BBP700': 'D', 'DOXY': ['A', 'D']})  # Take a␣
↪dict.
```

Search result properties and methods:

```
>>> idx.N_MATCH  # Shortcut for length of 1st dimension of the search results array
>>> idx.search  # Internal table with search results
>>> idx.uri  # List of absolute path to files from the search results table column
↪'file'
```

```
>>> idx.run()  # Run the search and save results in cache if necessary
>>> idx.to_dataframe()  # Convert search results to user-friendly :class:`pandas.
↪DataFrame`
>>> idx.to_dataframe(nrows=2)  # Only returns the first nrows of the search results
>>> idx.to_indexfile("search_index.txt")  # Export search results to Argo standard␣
↪index file
```

Misc:

```
>>> idx.convention  # What is the expected index format (core vs BGC profile index)
>>> idx.cname
>>> idx.read_wmo
>>> idx.read_params
>>> idx.records_per_wmo
```

__init__(*host: str = 'https://data-argo.ifremer.fr'*, *index_file: str = 'ar_index_global_prof.txt'*, *convention: str | None = None*, *cache: bool = False*, *cachedir: str = ''*, *timeout: int = 0*) → object

Create an Argo index file store

Parameters

- **host** (str, default: `https://data-argo.ifremer.fr`) – Local or remote (ftp or http) path to a *dac* folder (GDAC structure compliant). This takes values like: `ftp://ftp.ifremer.fr/ifremer/argo`, `ftp://usgodae.org/pub/outgoing/argo` or a local absolute path.
- **index_file** (str, default: `ar_index_global_prof.txt`) – Name of the csv-like text file with the index.

  Possible values are standard file name: `ar_index_global_prof.txt`, `argo_bio-profile_index.txt` or `argo_synthetic-profile_index.txt`.

  You can also use the following shortcuts: `core`, `bgc-b`, `bgc-s`, respectively.
- **convention** (*str, default: None*) –

  Set the expected format convention of the index file. This is useful when trying to load index file with custom name. If set to `None`, we'll try to infer the convention from the `index_file` value.

  Possible values: `ar_index_global_prof`, `argo_bio-profile_index`, or `argo_synthetic-profile_index`.

  You can also use the keyword: `core`, `bgc-s`, `bgc-b`.

- **cache** (*bool, default: False*) – Use cache or not.
- **cachedir** (*str, default: OPTIONS['cachedir']*) – Folder where to store cached files
- **timeout** (*int, default: OPTIONS['api_timeout']*) – Time out in seconds to connect to a remote host (ftp or http).

Methods

__init__([host, index_file, convention, ...])	Create an Argo index file store
cachepath(path)	Return path to a cached file
clear_cache()	Clear cache registry and files associated with this store instance.
load([nrows, force])	Load an Argo-index file content
read_params([index])	Return list of unique PARAMETERs in index or search results
read_wmo([index])	Return list of unique WMOs in search results
records_per_wmo([index])	Return the number of records per unique WMOs in search results
run([nrows])	Filter index with search criteria
search_cyc(CYCs[, nrows])	Search index for cycle numbers
search_lat_lon(BOX[, nrows])	Search index for a rectangular latitude/longitude domain
search_lat_lon_tim(BOX[, nrows])	Search index for a rectangular latitude/longitude domain and time range
search_parameter_data_mode(PARAMs[, ...])	Search index for profiles with a parameter in a specific data mode
search_params(PARAMs[, logical, nrows])	Search index for one or a list of parameters
search_tim(BOX[, nrows])	Search index for a time range
search_wmo(WMOs[, nrows])	Search index for floats defined by their WMO
search_wmo_cyc(WMOs, CYCs[, nrows])	Search index for floats defined by their WMO and specific cycle numbers
to_dataframe([nrows, index, completed])	Return index or search results as pandas.DataFrame
to_indexfile(outputfile)	Save search results on file, following the Argo standard index formats

Attributes

N_FILES	Number of rows in search result or index if search not triggered
N_MATCH	Number of rows in search result
N_RECORDS	Number of rows in the full index
backend	Name of store backend
cname	Return the search constraint(s) as a pretty formatted string
convention	Convention of the index (standard csv file name)
convention_supported	List of supported conventions
convention_title	Long name for the index convention
ext	Storage file extension
search_path	Path to search result uri
search_type	Dictionary with search meta-data
sha_df	Returns a unique SHA for a cname/dataframe
sha_h5	Returns a unique SHA for a cname/hdf5
sha_pq	Returns a unique SHA for a cname/parquet
shape	Shape of the index array
uri	List of URI from search results
uri_full_index	List of URI from index

argopy.ArgoNVSReferenceTables

class ArgoNVSReferenceTables(*nvs='https://vocab.nerc.ac.uk/collection'*, *cache: bool = True*, *cachedir: str = ''*)

Argo Reference Tables

Utility function to retrieve Argo Reference Tables from a NVS server.

By default, this relies on: https://vocab.nerc.ac.uk/collection

Examples

Methods:

```
>>> R = ArgoNVSReferenceTables()
>>> R.search('sensor')
>>> R.tbl(3)
>>> R.tbl('R09')
```

Properties:

```
>>> R.all_tbl_name
>>> R.all_tbl
>>> R.valid_ref
```

__init__(*nvs='https://vocab.nerc.ac.uk/collection'*, *cache: bool = True*, *cachedir: str = ''*)

Argo Reference Tables from NVS

Methods

`__init__`([nvs, cache, cachedir])	Argo Reference Tables from NVS
`get_url`(rtid[, fmt])	Return URL toward a given reference table for a given format
`search`(txt[, where])	Search for string in tables title and/or description
`tbl`(rtid)	Return an Argo Reference table
`tbl_name`(rtid)	Return name of an Argo Reference table

Attributes

`all_tbl`	Return all Argo Reference tables
`all_tbl_name`	Return names of all Argo Reference tables
`valid_ref`	List of all available Reference Tables

argopy.OceanOPSDeployments

class `OceanOPSDeployments`(*box: list | None = None*, *deployed_only: bool = False*)

Use the OceanOPS API for metadata access to retrieve Argo floats deployment information.

The API is documented here: https://www.ocean-ops.org/api/swagger/?url=https://www.ocean-ops.org/api/1/oceanops-api.yaml

Description of deployment status name:

Status	Id	Description
PROBABLE	0	Starting status for some platforms, when there is only a few metadata available, like rough deployment location and date. The platform may be deployed
CONFIRMED	1	Automatically set when a ship is attached to the deployment information. The platform is ready to be deployed, deployment is planned
REGISTERED	2	Starting status for most of the networks, when deployment planning is not done. The deployment is certain, and a notification has been sent via the OceanOPS system
OPERATIONAL	6	Automatically set when the platform is emitting a pulse and observations are distributed within a certain time interval
INACTIVE	4	The platform is not emitting a pulse since a certain time
CLOSED	5	The platform is not emitting a pulse since a long time, it is considered as dead

Examples

Import the class:

```
>>> from argopy.related import OceanOPSDeployments
>>> from argopy import OceanOPSDeployments
```

Possibly define the space/time box to work with:

```
>>> box = [-20, 0, 42, 51]
>>> box = [-20, 0, 42, 51, '2020-01', '2021-01']
>>> box = [-180, 180, -90, 90, '2020-01', None]
```

Instantiate the metadata fetcher:

```
>>> deployment = OceanOPSDeployments()
>>> deployment = OceanOPSDeployments(box)
>>> deployment = OceanOPSDeployments(box, deployed_only=True) # Remove planification
```

Load information:

```
>>> df = deployment.to_dataframe()
>>> data = deployment.to_json()
```

Useful attributes and methods:

```
>>> deployment.uri
>>> deployment.uri_decoded
>>> deployment.status_code
>>> fig, ax = deployment.plot_status()
>>> plan_virtualfleet = deployment.plan
```

__init__(*box: list | None = None*, *deployed_only: bool = False*)

Parameters

- **box** (*list, optional, default=None*) – Define the domain to load the Argo deployment plan for. By default, **box** is set to None to work with the global deployment plan starting from the current date. The list expects one of the following format:
  - [lon_min, lon_max, lat_min, lat_max]
  - [lon_min, lon_max, lat_min, lat_max, date_min]
  - [lon_min, lon_max, lat_min, lat_max, date_min, date_max]

  Longitude and latitude values must be floats. Dates are strings. If **box** is provided with a regional domain definition (only 4 values given), then `date_min` will be set to the current date.
- **deployed_only** (*bool, optional, default=False*) – Return only floats already deployed. If set to False (default), will return the full deployment plan (floats with all possible status). If set to True, will return only floats with one of the following status: `OPERATIONAL`, `INACTIVE`, and `CLOSED`.

Methods

`__init__`([box, deployed_only])	**param box** Define the domain to load the Argo deployment plan for. By default, **box** is set to None to work with the
`exp`([encoded])	Return an Ocean-Ops API deployment search expression for an argopy region box definition
`include`([encoded])	Return an Ocean-Ops API 'include' expression
`plot_status`(**kwargs)	Quick plot of the deployment plan
`to_dataframe`()	Return the deployment plan as `pandas.DataFrame`
`to_json`()	Return OceanOPS API request response as a json object

Attributes

`api`	URL to the API
`api_server_check`	URL to check if the API is alive
`box_name`	Return a string to print the box property
`model`	This model represents a Platform entity and is used to retrieve a platform information (schema model named 'Ptf').
`plan`	Return a dictionary to be used as argument in a `virtualargofleet.VirtualFleet`
`size`	
`status_code`	Return a `pandas.DataFrame` with the definition of status
`uri`	Return encoded URL to post an Ocean-Ops API request
`uri_decoded`	Return decoded URL to post an Ocean-Ops API request

argopy.CTDRefDataFetcher

CTDRefDataFetcher

alias of `Fetch_box`

argopy.TopoFetcher

class TopoFetcher(*box: list*, *ds: str = 'gebco'*, *cache: bool = False*, *cachedir: str = ''*, *api_timeout: int = 0*, *stride: list = [1, 1]*, *server: str | None = None*, ***kwargs*)

Fetch topographic data through an ERDDAP server for an ocean rectangle

Example

```
>>> from argopy import TopoFetcher
>>> box = [-75, -45, 20, 30]  # Lon_min, lon_max, lat_min, lat_max
>>> ds = TopoFetcher(box).to_xarray()
>>> ds = TopoFetcher(box, ds='gebco', stride=[10, 10], cache=True).to_xarray()
```

__init__(*box: list*, *ds: str = 'gebco'*, *cache: bool = False*, *cachedir: str = ''*, *api_timeout: int = 0*, *stride: list = [1, 1]*, *server: str | None = None*, ***kwargs*)

Instantiate an ERDDAP topo data fetcher

Parameters

- **ds** (*str (optional), default: 'gebco'*) – Dataset to load:
  - 'gebco' will load the GEBCO_2020 Grid, a continuous terrain model for oceans and land at 15 arc-second intervals
- **stride** (*list, default [1, 1]*) – Strides along longitude and latitude. This allows to change the output resolution
- **cache** (*bool (optional)*) – Cache data or not (default: False)
- **cachedir** (*str (optional)*) – Path to cache folder
- **api_timeout** (*int (optional)*) – Erddap request time out in seconds. Set to OPTIONS['api_timeout'] by default.

Methods

__init__(box[, ds, cache, cachedir, ...])	Instantiate an ERDDAP topo data fetcher
cname()	Return a unique string defining the constraints
define_constraints()	Define request constraints
get_url()	Return the URL to download data requested
load([errors])	Load Topographic data and return a xarray.DataSet
to_xarray([errors])	Load Topographic data and return a xarray.DataSet
url_encode(url)	Return safely encoded list of urls

Attributes

cachepath	Return path to cached file(s) for this request
uri	List of files to load for a request

argopy.ArgoDocs

class ArgoDocs(*docid=None*, *cache=False*)

ADMT documentation helper class

Examples

```
>>> ArgoDocs().list
>>> ArgoDocs().search("CDOM")
>>> ArgoDocs().search("CDOM", where='abstract')
```

```
>>> ArgoDocs(35385)
>>> ArgoDocs(35385).ris
>>> ArgoDocs(35385).abstract
>>> ArgoDocs(35385).show()
>>> ArgoDocs(35385).open_pdf()
>>> ArgoDocs(35385).open_pdf(page=12)
```

__init__(*docid=None*, *cache=False*)

Methods

__init__([docid, cache])	
open_pdf([page, url_only])	Open document in new browser tab
search(txt[, where])	Search for string in all documents title or abstract
show([height])	Insert document in pdf in a notebook cell

Attributes

abstract	Abstract of a document
js	Internal json record for a document
list	List of all available documents as a pandas.DataFrame
pdf	Link to the online pdf version of a document
ris	RIS record of a document

argopy.related.get_coriolis_profile_id

get_coriolis_profile_id(*WMO*, *CYC=None*, ***kwargs*)

Return a pandas.DataFrame with CORIOLIS ID of WMO/CYC profile pairs

This method get ID by requesting the dataselection.euro-argo.eu trajectory API.

Parameters

- **WMO** (*int, list(int)*) – Define the list of Argo floats. This is a list of integers with WMO float identifiers. WMO is the World Meteorological Organization.
- **CYC** (*int, list(int)*) – Define the list of cycle numbers to load ID for each Argo floats listed in WMO.

Return type
pandas.DataFrame

argopy.related.get_ea_profile_page

get_ea_profile_page(*WMO*, *CYC=None*, ***kwargs*)

Return a list of URL

Parameters

- **WMO** (*int, list(int)*) – WMO must be an integer or an iterable with elements that can be casted as integers
- **CYC** (*int, list(int), default (None)*) – CYC must be an integer or an iterable with elements that can be casted as positive integers

Return type
list(str)

See also:

get_coriolis_profile_id

1.12.3 Data visualisation

Visualisation functions available at the argopy module level:

dashboard([wmo, cyc, type, url_only, width, ...])	Insert an Argo dashboard page in a notebook cell, or return the corresponding url
ArgoColors([name, N])	Class to manage discrete coloring for Argo related variables

argopy.dashboard

dashboard(*wmo=None*, *cyc=None*, *type='ea'*, *url_only=False*, *width='100%'*, *height=1000*)

Insert an Argo dashboard page in a notebook cell, or return the corresponding url

Parameters

- **wmo** (*int, optional*) – The float WMO to display. By default, this is set to None and will insert the general dashboard.
- **cyc** (*int, optional*) – The float CYCLE NUMBER to display. If `wmo` is not None, this will open a profile dashboard.
- **type** (*str, optional, default: "ea"*) – Type of dashboard to use. This can be any one of the following:
  - "ea", "data": the Euro-Argo data selection dashboard
  - "meta": the Euro-Argo fleet monitoring dashboard
  - "op", "ocean-ops": the Ocean-OPS Argo dashboard
  - "bgc": the Argo-BGC specific dashboard
  - "argovis": the Colorado Argovis dashboard
- **url_only** (*bool, optional, default: False*) – If set to True, will only return the URL toward the dashboard
- **width** (*str, optional, default: "100%"*) – Width in percentage or pixel of the returned Iframe or Image
- **height** (*int, optional, default: 1000*) – Height in pixel of the returned Iframe or Image

Return type

str or `IPython.display.IFrame` or `IPython.display.Image`

Examples

Directly:

```
>>> argopy.dashboard()
>>> argopy.dashboard(6902745)
>>> argopy.dashboard(6902745, 12)
>>> argopy.dashboard(6902745, type='ocean-ops')
>>> argopy.dashboard(6902745, 12, url_only=True)
```

Or from a fetcher with the method `dashboard`:

```
>>> DataFetcher().float(6902745).dashboard()
```

argopy.ArgoColors

class ArgoColors(*name: str = 'Set1', N: int | None = None*)

Class to manage discrete coloring for Argo related variables

Call signatures:

```
from argopy.plot import ArgoColors

ArgoColors().list_valid_known_colormaps
ArgoColors().known_colormaps.keys()

ArgoColors('data_mode')
ArgoColors('data_mode').cmap
ArgoColors('data_mode').definition

ArgoColors('Set2').cmap
ArgoColors('Spectral', N=25).cmap
```

__init__(*name: str = 'Set1', N: int | None = None*)

Parameters

- **name** (*str, default: 'Set1'*) – Name of the colormap to use.
- **N** (*int, default: None*) – Number of colors to reduce the colormap to. If set to None, use the known quantitative colormap number of colors or fall back on a default 12 value.

Methods

__init__([name, N])	**param name** Name of the colormap to use.
cbar([ticklabels])	Return a colorbar with adjusted tick labels, **experimental**
show_COLORS()	Generate an HTML representation of the ArgoColors.COLORS palette
to_rgba(range, value)	Return the RGBA color for a given value of the colormap and a range

Attributes

COLORS	Set of Argo colors derived from the logo
cmap	Discrete colormap as matplotlib.colors.LinearSegmentedColormap
definition	Definition of the current known colormap, as a dictionary
list_valid_known_colormaps	List of all known colormaps, including alternative names
lookup	Dictionary with ticks as keys and colors as values
palette	Try to return a seaborn color palette as a list of RGB tuples or matplotlib.colors.ListedColormap
quantitative	Dictionary with number of colors in known quantitative maps
ticklabels	Dictionary with ticks as keys and ticklabels as values

All other visualisation functions are in the `argopy.plot` submodule:

open_sat_altim_report([WMO, embed])	Insert the CLS Satellite Altimeter Report figure in notebook cell
scatter_map(data[, x, y, hue, markersize, ...])	Try-to-be generic function to create a scatter plot on a map from **argopy** xarray.Dataset or pandas.DataFrame data
bar_plot(df[, by, style, with_seaborn])	Create a bar plot for an Argo index dataframe
scatter_plot(ds, this_param[, this_x, ...])	A quick-and-dirty parameter scatter plot for one variable
latlongrid(ax[, dx, dy, fontsize, ...])	Add latitude/longitude grid line and labels to a cartopy geoaxes

argopy.plot.open_sat_altim_report

open_sat_altim_report(*WMO: str | list | None = None*, *embed: str | None = 'dropdown'*, ***kwargs*)

Insert the CLS Satellite Altimeter Report figure in notebook cell

This is the method called when using the facade fetcher methods `plot`:

```
DataFetcher().float(6902745).plot('qc_altimetry')
```

Parameters

- **WMO** (*int or list*) – The float WMO to display. By default, this is set to None and will insert the general dashboard.
- **embed** (*str, default='dropdown'*) – Set the embedding method. If set to None, simply return the list of urls to figures. Possible values are: `dropdown`, `slide` and `list`.

Return type

list of Image with `list` embed or a dict with URLs

Notes

Requires IPython to work as expected. If IPython is not available only URLs are returned.

argopy.plot.scatter_map

scatter_map(*data: Dataset | DataFrame*, *x: str | None = None*, *y: str | None = None*, *hue: str | None = None*, *markersize: int = 36*, *markeredgesize: float = 0.5*, *markeredgecolor: str = 'default'*, *cmap: str | None = None*, *traj: bool = True*, *traj_axis: str | None = None*, *traj_color: str = 'default'*, *legend: bool = True*, *legend_title: str = 'default'*, *legend_location: str | int = 0*, *cbar: bool = False*, *cbarlabels: str | list = 'auto'*, *set_global: bool = False*, ***kwargs*)

Try-to-be generic function to create a scatter plot on a map from **argopy** `xarray.Dataset` or `pandas.DataFrame` data

Each point is an Argo profile location, colored with a user defined variable and colormap. Floats trajectory can be plotted or not.

Note that all parameters have default values.

Warning: This function requires Cartopy.

Examples

```
from argopy.plot import scatter_map
from argopy import DataFetcher

ArgoSet = DataFetcher(mode='expert').float([6902771, 4903348]).load()
ds = ArgoSet.data.argo.point2profile()
df = ArgoSet.index

scatter_map(df)
scatter_map(ds)
scatter_map(ds, hue='DATA_MODE')
scatter_map(ds, hue='PSAL_QC')
```

```
from argopy import OceanOPSDeployments
df = OceanOPSDeployments([-90, 0, 0, 90]).to_dataframe()
scatter_map(df, hue='status_code', traj=False)
scatter_map(df, x='lon', y='lat', hue='status_code', traj=False, cmap='deployment_
↪status')
```

Parameters

- **data** (`xarray.Dataset` or `pandas.DataFrame`) – Input data structure
- **x** (*str, default=None*) – Name of the data variable to use as longitude. If x is set to None, we'll try to guess which variable to use among standard names.
- **y** (*str, default=None*) – Name of the data variable to use as latitude. If y is set to None, we'll try to guess which variable to use among standard names.
- **hue** (*str, default=None*) – Name of the data variable to use for points coloring. If hue is set to None, we'll try to guess which variable to use to color points according to WMO.

- **markersize** (*int, default=36*) – Size of the marker used for profiles location.
- **markeredgesize** (*float, default=0.5*) – Size of the marker edge used for profiles location.
- **markeredgecolor** (*str, default='default'*) – Color to use for the markers edge. The default color is 'DARKBLUE' from `argopy.plot.ArgoColors.COLORS`
- **cmap** (*str, default=None*) – Colormap to use for points coloring. If set to None, we'll try to guess the most appropriate colormap for the `hue` argument by matching it to values in `argopy.plot.ArgoColors.list_valid_known_colormaps`.
- **traj** (*bool, default=True*) – Set to True in order to plot each float trajectories, i.e. join with a line all profiles from a single platform.
- **traj_axis** (*str, default='wmo'*) – Name of the data variable to use in order to determine profiles group making a single trajectory.
- **traj_color** (*str, default='default'*) – The unique color to use for all trajectories. The default color is the `markeredgecolor` value.
- **legend** (*bool, default=True*) – Display or not a legend for hue colors meaning. If the legend is too large, it can be removed with `ax.get_legend().remove()`
- **legend_title** (*str, default='default'*) – String title of the legend box. By default, it is set to the `hue` value.
- **legend_location** (*str, default='upper right'*) – Location of the legend box. This is passed to the `loc` argument of `Legend`.
- **set_global** (*bool, default=False*) – Force the map to be global.
- **kwargs** – All other arguments are passed to `matplotlib.figure.Figure.subplots`

Returns

- **fig** (`matplotlib.figure.Figure`)
- **ax** (`matplotlib.axes.Axes`)

argopy.plot.bar_plot

bar_plot(*df: DataFrame*, *by: str = 'institution'*, *style: str = 'whitegrid'*, *with_seaborn: bool = False*, ***kwargs*)

Create a bar plot for an Argo index dataframe

This is the method called when using the facade fetcher methods `plot` with the `dac` or `profiler` arguments:

```
IndexFetcher(src='gdac').region([-80,-30,20,50,'2021-01','2021-08']).plot('dac')
```

To use it directly, you must pass a `pandas.DataFrame` as returned by a *argopy.DataFetcher.index* or *argopy.IndexFetcher.index* property:

```
from argopy import IndexFetcher
df = IndexFetcher(src='gdac').region([-80,-30,20,50,'2021-01','2021-08']).index
bar_plot(df, by='profiler')
```

Parameters

- **df** (`pandas.DataFrame`) – As returned by a fetcher index property
- **by** (*str, default='institution'*) – The profile property to plot

- **style** (*str, optional*) – Define the Seaborn axes style: ‘white’, ‘darkgrid’, ‘whitegrid’, ‘dark’, ‘ticks’

Returns

- **fig** (matplotlib.figure.Figure)
- **ax** (matplotlib.axes.Axes)

argopy.plot.scatter_plot

scatter_plot(*ds: Dataset*, *this_param*, *this_x='TIME'*, *this_y='PRES'*, *figsize=(18, 6)*, *cmap=None*, *vmin=None*, *vmax=None*, *s=4*, *bgcolor='lightgrey'*)

A quick-and-dirty parameter scatter plot for one variable

argopy.plot.latlongrid

latlongrid(*ax*, *dx='auto'*, *dy='auto'*, *fontsize='auto'*, *label_style_arg={}*, ***kwargs*)

Add latitude/longitude grid line and labels to a cartopy geoaxes

Parameters

- **ax** (*cartopy.mpl.geoaxes.GeoAxesSubplot*) – Cartopy axes to add the lat/lon grid to
- **dx** (*'auto' or float*) – Grid spacing along longitude
- **dy** (*'auto' or float*) – Grid spacing along latitude
- **fontsize** (*'auto' or int*) – Grid label font size

Returns
class

Return type
cartopy.mpl.geoaxes.GeoAxesSubplot.gridlines

1.12.4 Dataset.argo (xarray accessor)

Dataset.argo	Class registered under scope argo to access a xarray.Dataset object.

xarray.Dataset.argo

Dataset.**argo**()

Class registered under scope argo to access a xarray.Dataset object.

Examples

- Ensure all variables are of the Argo required dtype with:

```
>>> ds.argo.cast_types()
- Convert a collection of points into a collection of profiles:
>>> ds.argo.point2profile()
- Convert a collection of profiles to a collection of points:
>>> ds.argo.profile2point()
- Filter measurements according to data mode:
>>> ds.argo.filter_data_mode()
- Filter measurements according to QC flag values:
>>> ds.argo.filter_qc(QC_list=[1, 2], QC_fields='all')
- Filter variables according OWC salinity calibration requirements:
>>> ds.argo.filter_scalib_pres(force='default')
- Interpolate measurements on pressure levels:
>>> ds.argo.inter_std_levels(std_lev=[10., 500., 1000.])
- Group and reduce measurements by pressure bins:
>>> ds.argo.groupby_pressure_bins(bins=[0, 200., 500., 1000.])
- Compute and add additional variables to the dataset:
>>> ds.argo.teos10(vlist='PV')
- Preprocess data for OWC salinity calibration:
>>> ds.argo.create_float_source("output_folder")
```

This accessor extends `xarray.Dataset`. Proper use of this accessor should be like:

```
>>> import xarray as xr         # first import xarray
>>> import argopy               # import argopy (the dataset 'argo' accessor is␣
↪registered)
>>> from argopy import DataFetcher
>>> ds = DataFetcher().float([6902766, 6902772, 6902914, 6902746]).load().data
>>> ds.argo
>>> ds.argo.filter_qc()
```

Data Transformation

Dataset.argo.point2profile([drop])	Transform a collection of points into a collection of profiles
Dataset.argo.profile2point()	Convert a collection of profiles to a collection of points
Dataset.argo.interp_std_levels(std_lev[, axis])	Interpolate measurements to standard pressure levels
Dataset.argo.groupby_pressure_bins(bins[, ...])	Group measurements by pressure bins

xarray.Dataset.argo.point2profile

Dataset.argo.**point2profile**(*drop: bool = False*)

Transform a collection of points into a collection of profiles

A "point" is a single location for measurements in space and time A "point" is localised as unique UID based on WMO, CYCLE_NUMBER and DIRECTION variable values.

Parameters

drop (*bool, default=False*) – By default will return all variables. But if set to True, then all [N_PROF, N_LEVELS] 2d variables will be dropped, and only 1d variables of dimension [N_PROF] will be returned.

xarray.Dataset.argo.profile2point

Dataset.argo.**profile2point**()

Convert a collection of profiles to a collection of points

A "point" is a single location for measurements in space and time A "point" is localised as unique UID based on WMO, CYCLE_NUMBER and DIRECTION variable values.

xarray.Dataset.argo.interp_std_levels

Dataset.argo.**interp_std_levels**(*std_lev: list*, *axis: str = 'PRES'*)

Interpolate measurements to standard pressure levels

Parameters

- **std_lev** (*list or np.array*) – Standard pressure levels used for interpolation. It has to be 1-dimensional and monotonic.
- **axis** (str, default: PRES) – The dataset variable to use as pressure axis. This could be PRES or PRES_ADJUSTED.

Return type

xarray.Dataset

xarray.Dataset.argo.groupby_pressure_bins

Dataset.argo.**groupby_pressure_bins**(*bins: list*, *axis: str = 'PRES'*, *right: bool = False*, *select: str = 'deep'*, *squeeze: bool = True*, *merge: bool = True*)

Group measurements by pressure bins

This method can be used to subsample and align an irregular dataset (pressure not being similar in all profiles) on a set of pressure bins. The output dataset could then be used to perform statistics along the N_PROF dimension because N_LEVELS will correspond to similar pressure bins, while avoiding to interpolate data.

Parameters

- **bins** (*list or np.array,*) – Array of bins. It has to be 1-dimensional and monotonic. Bins of data are localised using values from options *axis* (default: PRES) and *right* (default: False), see below.
- **axis** (str, default: PRES) – The dataset variable to use as pressure axis. This could be PRES or PRES_ADJUSTED

- **right** (*bool, default: False*) – Indicating whether the bin intervals include the right or the left bin edge. Default behavior is (right==False) indicating that the interval does not include the right edge. The left bin end is open in this case, i.e., bins[i-1] <= x < bins[i] is the default behavior for monotonically increasing bins. Note the `merge` option is intended to work only for the default `right=False`.
- **select** (*{'deep','shallow','middle','random','min','max','mean','median'}, default: 'deep'*) – The value selection method for bins.

  This selection can be based on values at the pressure axis level with: `deep` (default), `shallow`, `middle`, `random`. For instance, `select='deep'` will lead to the value returned for a bin to be taken at the deepest pressure level in the bin.

  Or this selection can be based on statistics of measurements in a bin. Stats available are: `min`, `max`, `mean`, `median`. For instance `select='mean'` will lead to the value returned for a bin to be the mean of all measurements in the bin.
- **squeeze** (*bool, default: True*) – Squeeze from the output bin levels without measurements.
- **merge** (*bool, default: True*) – Optimize the output bins axis size by merging levels with/without data. The pressure bins axis is modified accordingly. This means that the return `STD_PRES_BINS` axis has not necessarily the same size as the input `bins`.

Return type
`xarray.Dataset`

See also:

`numpy.digitize`, `argopy.utilities.groupby_remap`

Data Filters

Dataset.argo.filter_qc([QC_list, QC_fields, ...])	Filter data set according to QC values
Dataset.argo.filter_data_mode([keep_error, ...])	Filter variables according to their data mode
Dataset.argo.filter_scalib_pres([force, inplace])	Filter variables according to OWC salinity calibration software requirements
Dataset.argo.filter_researchmode()	Filter dataset for research user mode

xarray.Dataset.argo.filter_qc

`Dataset.argo.`**`filter_qc`**(*QC_list=[1, 2]*, *QC_fields='all'*, *drop=True*, *mode='all'*, *mask=False*)

Filter data set according to QC values

Filter the dataset to keep points where `all` or `any` of the QC fields has a value in the list of integer QC flags.

This method can return the filtered dataset or the filter mask.

Parameters

- **QC_list** (*list(int)*) – List of QC flag values (integers) to keep
- **QC_fields** (*'all' or list(str)*) – List of QC fields to consider to apply the filter. By default we use all available QC fields
- **drop** (*bool*) – Drop values not matching the QC filter, default is True

- **mode** (*str*) – Must be `all` (default) or `any`. Boolean operator on QC values: should we keep points matching `all` QC fields or 'any' one of them.
- **mask** (*bool*) – `False` by default. Determine if we should return the QC mask or the filtered dataset.

Return type
`xarray.Dataset`

xarray.Dataset.argo.filter_data_mode

`Dataset.argo.filter_data_mode(keep_error: bool = True, errors: str = 'raise')`

Filter variables according to their data mode

This filter applies to <PARAM> and <PARAM_QC>

For data mode 'R' and 'A': keep <PARAM> (eg: 'PRES', 'TEMP' and 'PSAL')

For data mode 'D': keep <PARAM_ADJUSTED> (eg: 'PRES_ADJUSTED', 'TEMP_ADJUSTED' and 'PSAL_ADJUSTED')

Since ADJUSTED variables are not required anymore after the filter, all *ADJUSTED* variables are dropped in order to avoid confusion wrt variable content. DATA_MODE is preserved for the record.

Parameters

- **keep_error** (*bool, optional*) – If true (default) keep the measurements error fields or not.
- **errors** (*{'raise','ignore'}, optional*) – If 'raise' (default), raises a InvalidDatasetStructure error if any of the expected dataset variables is not found. If 'ignore', fails silently and return unmodified dataset.

Return type
`xarray.Dataset`

xarray.Dataset.argo.filter_scalib_pres

`Dataset.argo.filter_scalib_pres(force: str = 'default', inplace: bool = True)`

Filter variables according to OWC salinity calibration software requirements

By default, this filter will return a dataset with raw PRES, PSAL and TEMP; and if PRES is adjusted, PRES variable will be replaced by PRES_ADJUSTED.

With option force='raw', you can force the filter to return a dataset with raw PRES, PSAL and TEMP whether PRES is adjusted or not.

With option force='adjusted', you can force the filter to return a dataset where PRES/PSAL and TEMP replaced with adjusted variables: PRES_ADJUSTED, PSAL_ADJUSTED, TEMP_ADJUSTED.

Since ADJUSTED variables are not required anymore after the filter, all *ADJUSTED* variables are dropped in order to avoid confusion wrt variable content.

Parameters

- **force** (*str*) – Use force='default' to load PRES/PSAL/TEMP or PRES_ADJUSTED/PSAL/TEMP according to PRES_ADJUSTED filled or not.

  Use force='raw' to force load of PRES/PSAL/TEMP

  Use force='adjusted' to force load of PRES_ADJUSTED/PSAL_ADJUSTED/TEMP_ADJUSTED

- **inplace** (*boolean, True by default*) – If True, return the filtered input xarray.Dataset

  If False, return a new xarray.Dataset

Return type
xarray.Dataset

xarray.Dataset.argo.filter_researchmode

Dataset.argo.**filter_researchmode**() → Dataset

Filter dataset for research user mode

This filter will select only data with QC=1, in delayed mode and with pressure errors smaller than 20db

Return type
xarray.Dataset

Processing

Dataset.argo.teos10([vlist, inplace])	Add TEOS10 variables to the dataset
Dataset.argo.create_float_source([path, ...])	Preprocess data for OWC software calibration

xarray.Dataset.argo.teos10

Dataset.argo.**teos10**(*vlist: list = ['SA', 'CT', 'SIG0', 'N2', 'PV', 'PTEMP']*, *inplace: bool = True*)

Add TEOS10 variables to the dataset

By default, adds: 'SA', 'CT' Other possible variables: 'SIG0', 'N2', 'PV', 'PTEMP', 'SOUND_SPEED' Relies on the gsw library.

If one exists, the correct CF standard name will be added to the attrs.

Parameters

- **vlist** (*list(str)*) – List with the name of variables to add. Must be a list containing one or more of the following string values:
  - **SA**
    Adds an absolute salinity variable
  - **CT**
    Adds a conservative temperature variable
  - **SIG0**
    Adds a potential density anomaly variable referenced to 0 dbar
  - **N2**
    Adds a buoyancy (Brunt-Vaisala) frequency squared variable. This variable has been regridded to the original pressure levels in the Dataset using a linear interpolation.
  - **PV**
    Adds a planetary vorticity variable calculated from $\frac{fN^2}{\text{gravity}}$. This is not a TEOS-10 variable from the gsw toolbox, but is provided for convenience. This variable has been regridded to the original pressure levels in the Dataset using a linear interpolation.

- **PTEMP**
  Add potential temperature
- **SOUND_SPEED**
  Add sound speed
- **CNDC**
  Add Electrical Conductivity

• **inplace** (*boolean, True by default*) –

- **If True, return the input `xarray.Dataset` with new TEOS10 variables**
  added as a new `xarray.DataArray`.
- If False, return a `xarray.Dataset` with new TEOS10 variables

Return type
`xarray.Dataset`

xarray.Dataset.argo.create_float_source

Dataset.argo.**create_float_source**(*path: str | None = None*, *force: str = 'default'*, *select: str = 'deep'*, *file_pref: str = ''*, *file_suff: str = ''*, *format: str = '5'*, *do_compression: bool = True*, *debug_output: bool = False*)

Preprocess data for OWC software calibration

This method can create a FLOAT SOURCE file (i.e. the .mat file that usually goes into /float_source/) for OWC software. The FLOAT SOURCE file is saved as:

<path>/<file_pref><float_WMO><file_suff>.mat

where <float_WMO> is automatically extracted from the dataset variable PLATFORM_NUMBER (in order to avoid mismatch between user input and data content). So if this dataset has measurements from more than one float, more than one Matlab file will be created.

By default, variables loaded are raw PRES, PSAL and TEMP. If PRES is adjusted, variables loaded are PRES_ADJUSTED, raw PSAL calibrated in pressure and raw TEMP.

You can force the program to load raw PRES, PSAL and TEMP whatever PRES is adjusted or not:

```
>>> ds.argo.create_float_source(force='raw')
```

or you can force the program to load adjusted variables: PRES_ADJUSTED, PSAL_ADJUSTED, TEMP_ADJUSTED

```
>>> ds.argo.create_float_source(force='adjusted')
```

Pre-processing details:

1. select only ascending profiles
2. subsample vertical levels to keep the deepest pressure levels on each 10db bins from the surface down to the deepest level.
3. align pressure values, i.e. make sure that a pressure index corresponds to measurements from the same binned pressure values. This can lead to modify the number of levels in the dataset.
4. filter variables according to the `force` option (see below)
5. filter variables according to QC flags:
   - Remove measurements where timestamp QC is >= 3

- Keep measurements where pressure QC is anything but 3
- Keep measurements where pressure, temperature or salinity QC are anything but 4

6. remove dummy values: salinity not in [0/50], potential temperature not in [-10/50] and pressure not in [0/60000]. Bounds inclusive.
7. convert timestamp to fractional year
8. convert longitudes to 0-360

Parameters

- **path** (*str or path-like, optional*) – Path or folder name to which to save this Matlab file. If no path is provided, this function returns the resulting Matlab file as `xarray.Dataset`.
- **force** (*{"default", "raw", "adjusted"}, default: "default"*) – If force='default' will load PRES/PSAL/TEMP or PRES_ADJUSTED/PSAL/TEMP according to PRES_ADJUSTED filled or not.

  If force='raw' will load PRES/PSAL/TEMP

  If force='adjusted' will load PRES_ADJUSTED/PSAL_ADJUSTED/TEMP_ADJUSTED
- **select** (*{'deep','shallow','middle','random','min','max','mean','median'}, default: 'deep'*) –
- **file_pref** (*str, optional*) – Prefix to add at the beginning of output file(s).
- **file_suff** (*str, optional*) – Suffix to add at the end of output file(s).
- **do_compression** (*bool, optional*) – Whether to compress matrices on write. Default is True.
- **format** (*{'5', '4'}, string, optional*) – Matlab file format version. '5' (the default) for MATLAB 5 and up (to 7.2). Use '4' for MATLAB 4 .mat files.

Returns

The output dataset, or Matlab file, will have the following variables (`n` is the number of profiles, `m` is the number of vertical levels):

- `DATES` (1xn): decimal year, e.g. 10 Dec 2000 = 2000.939726
- `LAT` (1xn): decimal degrees, -ve means south of the equator, e.g. 20.5S = -20.5
- `LONG` (1xn): decimal degrees, from 0 to 360, e.g. 98.5W in the eastern Pacific = 261.5E
- `PROFILE_NO` (1xn): this goes from 1 to n. PROFILE_NO is the same as CYCLE_NO in the Argo files
- `PRES` (mxn): dbar, from shallow to deep, e.g. 10, 20, 30 … These have to line up along a fixed nominal depth axis.
- `TEMP` (mxn): in-situ IPTS-90
- `SAL` (mxn): PSS-78
- `PTMP` (mxn): potential temperature referenced to zero pressure, use SAL in PSS-78 and in-situ TEMP in IPTS-90 for calculation.

Return type
`xarray.Dataset`

Misc

Dataset.argo.index	Basic profile index
Dataset.argo.domain	Space/time domain of the dataset
Dataset.argo.list_WMO_CYC()	Given a dataset, return a list with all possible (PLATFORM_NUMBER, CYCLE_NUMBER) tuple
Dataset.argo.uid(wmo_or_uid[, cyc, direction])	UID encoder/decoder
Dataset.argo.cast_types(**kwargs)	Make sure variables are of the appropriate types according to Argo

xarray.Dataset.argo.index

Dataset.argo.**index**()

Basic profile index

xarray.Dataset.argo.domain

Dataset.argo.**domain**()

Space/time domain of the dataset

This is different from a usual argopy `box` because dates are in `numpy.datetime64` format.

xarray.Dataset.argo.list_WMO_CYC

Dataset.argo.**list_WMO_CYC**()

Given a dataset, return a list with all possible (PLATFORM_NUMBER, CYCLE_NUMBER) tuple

xarray.Dataset.argo.uid

Dataset.argo.**uid**(*wmo_or_uid*, *cyc=None*, *direction=None*)

UID encoder/decoder

Parameters

- **int** – WMO number (to encode) or UID (to decode)
- **cyc** (*int, optional*) – Cycle number (to encode), not used to decode
- **direction** (*str, optional*) – Direction of the profile, must be 'A' (Ascending) or 'D' (Descending)

Return type

int or tuple of int

Examples

```
>>> unique_float_profile_id = uid(690024,13,'A') # Encode
>>> wmo, cyc, drc = uid(unique_float_profile_id) # Decode
```

xarray.Dataset.argo.cast_types

Dataset.argo.**cast_types**(***kwargs*)

Make sure variables are of the appropriate types according to Argo

1.12.5 Utilities

Function under the `argopy.utils` submodule.

list_available_data_src()	List all available data sources
list_available_index_src()	List all available index sources
list_standard_variables()	List of variables for standard users
list_multiprofile_file_variables()	List of variables in a netcdf multiprofile file.
check_wmo(lst[, errors])	Validate a WMO option and returned it as a list of integers
check_cyc(lst[, errors])	Validate a CYC option and returned it as a list of integers
float_wmo(WMO_number[, errors])	Argo float WMO number object
Registry([initlist, name, dtype, invalid])	A list manager can that validate item type
Chunker(request[, chunks, chunksize])	To chunk fetcher requests
isconnected([host, maxtry])	Check if an URL is alive
urlhaskeyword([url, keyword, maxtry])	Check if a keyword is in the content of a URL
isalive([api_server_check])	Check if an API is alive or not
isAPIconnected([src, data])	Check if a source API is alive or not
drop_variables_not_in_all_datasets(ds_collectic	Drop variables that are not in all datasets (the lowest common denominator)
fill_variables_not_in_all_datasets(ds_collectic	Add empty variables to dataset so that all the collection have the same `xarray.Dataset.data_vars` and **:props:`xarray.Dataset.coords`**

argopy.utils.list_available_data_src

list_available_data_src()

List all available data sources

argopy.utils.list_available_index_src

list_available_index_src()

List all available index sources

argopy.utils.list_standard_variables

list_standard_variables()

List of variables for standard users

argopy.utils.list_multiprofile_file_variables

list_multiprofile_file_variables()

List of variables in a netcdf multiprofile file.

This is for files created by GDAC under <DAC>/<WMO>/<WMO>_prof.nc

argopy.utils.check_wmo

check_wmo(*lst*, *errors='raise'*)

Validate a WMO option and returned it as a list of integers

Parameters

- **wmo** (*int*) – WMO must be an integer or an iterable with elements that can be casted as integers
- **errors** (*{'raise', 'warn', 'ignore'}*) – Possibly raises a ValueError exception or UserWarning, otherwise fails silently.

Return type
list(int)

argopy.utils.check_cyc

check_cyc(*lst*, *errors='raise'*)

Validate a CYC option and returned it as a list of integers

Parameters

- **cyc** (*int*) – CYC must be an integer or an iterable with elements that can be casted as positive integers
- **errors** (*{'raise', 'warn', 'ignore'}*) – Possibly raises a ValueError exception or UserWarning, otherwise fails silently.

Return type
list(int)

argopy.utils.float_wmo

class float_wmo(*WMO_number*, *errors='raise'*)

Argo float WMO number object

__init__(*WMO_number*, *errors='raise'*)

Create an Argo float WMO number object

Parameters

- **WMO_number** (*object*) – Anything that could be casted as an integer
- **errors** (*{'raise', 'warn', 'ignore'}*) – Possibly raises a ValueError exception or UserWarning, otherwise fails silently if WMO_number is not valid

Return type

argopy.utilities.float_wmo

Methods

__init__(WMO_number[, errors])	Create an Argo float WMO number object

Attributes

isvalid	Check if WMO number is valid
value	Return WMO number as in integer

argopy.utils.Registry

class Registry(*initlist=None*, *name: str = 'unnamed'*, *dtype='str'*, *invalid='raise'*)

A list manager can that validate item type

Examples

You can commit new entry to the registry, one by one:

```
>>> R = Registry(name='file')
>>> R.commit('meds/4901105/profiles/D4901105_017.nc')
>>> R.commit('aoml/1900046/profiles/D1900046_179.nc')
```

Or with a list:

```
>>> R = Registry(name='My floats', dtype='wmo')
>>> R.commit([2901746, 4902252])
```

And also at instantiation time (name and dtype are optional):

```
>>> R = Registry([2901746, 4902252], name='My floats', dtype=float_wmo)
```

Registry can be used like a list.

It is iterable:

```
>>> for wmo in R:
>>>     print(wmo)
```

It has a `len` property:

```
>>> len(R)
```

It can be checked for values:

```
>>> 4902252 in R
```

You can also remove items from the registry, again one by one or with a list:

```
>>> R.remove('2901746')
```

__init__(*initlist=None*, *name: str = 'unnamed'*, *dtype='str'*, *invalid='raise'*)

Create a registry, i.e. a controlled list

Parameters

- **initlist** (*list, optional*) – List of values to register
- **name** (*str, default: 'unnamed'*) – Name of the Registry
- **dtype** (`str` or dtype, default: `str`) – Data type of registry content. Supported values are: 'str', 'wmo', float_wmo
- **invalid** (*str, default: 'raise'*) – Define what do to when a new item is not valid. Can be 'raise' or 'ignore'

Methods

`__init__`([initlist, name, dtype, invalid])	Create a registry, i.e. a controlled list.
`append`(value)	R.append(value) -- append value to the end of the registry
`clear`()	
`commit`(values)	R.commit(values) -- append values to the end of the registry if not already in
`copy`()	Return a shallow copy of the registry
`count`(value)	
`extend`(other)	R.extend(iterable) -- extend registry by appending elements from the iterable
`index`(value, [start, [stop]])	Raises ValueError if the value is not present.
`insert`(index, value)	R.insert(index, value) -- insert value before index.
`pop`([index])	Raise IndexError if list is empty or index is out of range.
`remove`(values)	R.remove(valueS) -- remove first occurrence of values.
`reverse`()	S.reverse() -- reverse *IN PLACE*
`sort`(*args, **kwds)	

argopy.utils.Chunker

class Chunker(*request: dict*, *chunks: str = 'auto'*, *chunksize: dict = {}*)

To chunk fetcher requests

__init__(*request: dict*, *chunks: str = 'auto'*, *chunksize: dict = {}*)

Create a request Chunker

Allow to easily split an access point request into chunks

Parameters

- **request** (*dict*) – Access point request to be chunked. One of the following:
  - {'box': [lon_min, lon_max, lat_min, lat_max, dpt_min, dpt_max, time_min, time_max]}
  - {'box': [lon_min, lon_max, lat_min, lat_max, dpt_min, dpt_max]}
  - {'wmo': [wmo1, wmo2, ...], 'cyc': [0,1, ...]}
- **chunks** (*'auto' or dict*) – Dictionary with request access point as keys and number of chunks to create as values.

  Eg: {'wmo':10} will create a maximum of 10 chunks along WMOs.
- **chunksize** (*dict, optional*) – Dictionary with request access point as keys and chunk size as values (used as maximum values in 'auto' chunking).

  Eg: {'wmo': 5} will create chunks with as many as 5 WMOs each.

Methods

`__init__`(request[, chunks, chunksize])	Create a request Chunker
`fit_transform`()	Chunk a fetcher request

Attributes

`default_chunksize`

argopy.utils.isconnected

isconnected(*host: str = 'https://www.ifremer.fr'*, *maxtry: int = 10*)

Check if an URL is alive

Parameters

- **host** (*str*) – URL to use, 'https://www.ifremer.fr' by default
- **maxtry** (*int, default: 10*) – Maximum number of host connections to try before

Return type
bool

argopy.utils.urlhaskeyword

urlhaskeyword(*url: str = ''*, *keyword: str = ''*, *maxtry: int = 10*)

Check if a keyword is in the content of a URL

Parameters

- **url** (*str*) –
- **keyword** (*str*) –
- **maxtry** (*int, default: 10*) – Maximum number of host connections to try before returning False

Return type
bool

argopy.utils.isalive

isalive(*api_server_check: str | dict = ''*) → bool

Check if an API is alive or not

2 methods are available:

- URL Ping
- keyword Check

Parameters
api_server_check – Url string or dictionary with [`url`, `keyword`] keys.

- For a string, uses: `argopy.utilities.isconnected`
- For a dictionary, uses: `argopy.utilities.urlhaskeyword`

Return type
bool

argopy.utils.isAPIconnected

isAPIconnected(*src='erddap'*, *data=True*)

Check if a source API is alive or not

The API is connected when it has a live URL or valid folder path.

Parameters

- **src** (*str*) – The data or index source name, 'erddap' default
- **data** (*bool*) – If True check the data fetcher (default), if False, check the index fetcher

Return type
bool

argopy.utils.drop_variables_not_in_all_datasets

drop_variables_not_in_all_datasets(*ds_collection: List[Dataset]*) → List[Dataset]

Drop variables that are not in all datasets (the lowest common denominator)

Parameters
ds_collection (*List[xarray.Dataset]*) – A list of `xarray.Dataset`

Return type
List[xarray.Dataset]

argopy.utils.fill_variables_not_in_all_datasets

fill_variables_not_in_all_datasets(*ds_collection: List[Dataset]*, *concat_dim: str = 'rows'*) → List[Dataset]

Add empty variables to dataset so that all the collection have the same `xarray.Dataset.data_vars` and **:props:`xarray.Dataset.coords`**

This is to make sure that the collection of dataset can be concatenated

Parameters

- **ds_collection** (*List[xarray.Dataset]*) – A list of `xarray.Dataset`
- **concat_dim** (*str, default='rows'*) – Name of the dimension to use to create new variables. Typically, this is the name of the dimension the collection will be concatenated along afterward.

Return type
List[xarray.Dataset]

1.12.6 Argopy helpers

set_options(**kwargs)	Set options for argopy
clear_cache([fs])	Delete argopy cache folder content
tutorial.open_dataset(name)	Open a dataset from the argopy online data repository (requires internet).
show_versions([file, conda])	Print the versions of argopy and its dependencies
xarray.ArgoEngine()	Backend for Argo netCDF files based on the xarray netCDF4 engine

argopy.set_options

class set_options(***kwargs*)

Set options for argopy

List of options:

- **dataset: Define the Dataset to work with.**
  Default: phy. Possible values: phy, bgc or ref.
- **src: Source of fetched data.**
  Default: erddap. Possible values: erddap, gdac, argovis
- **mode: User mode.**
  Default: standard. Possible values: standard, expert or research.
- **ftp: Default path to be used by the GDAC fetchers and Argo index stores**
  Default: https://data-argo.ifremer.fr
- **erddap: Default server address to be used by the data and index erddap fetchers**
  Default: https://erddap.ifremer.fr/erddap
- **cachedir: Absolute path to a local cache directory.**
  Default: ~/.cache/argopy
- **cache_expiration: Expiration delay of cache files in seconds.**
  Default: 86400
- **api_timeout: Define the time out of internet requests to web API, in seconds.**
  Default: 60
- **trust_env: Allow for local environment variables to be used to connect to the internet.**
  Default: False. Argopy will get proxies information from HTTP_PROXY / HTTPS_PROXY environment variables if this option is True and it can also get proxy credentials from ~/.netrc file if this file exists.
- **user/password: Username and password to use when a simple authentication is required.**
  Default: None, None
- **server: Other than expected/default server to be uses by a function/method. This is mostly intended to be used for unit testing**
  Default: None

You can use set_options either as a context manager for temporary setting:

```
>>> import argopy
>>> with argopy.set_options(src='gdac'):
>>>     ds = argopy.DataFetcher().float(3901530).to_xarray()
```

or to set global options (at the beginning of a script for instance):

```
>>> argopy.set_options(src='gdac')
```

__init__(***kwargs*)

Methods

__init__(**kwargs)

argopy.clear_cache

clear_cache(*fs=None*)

Delete argopy cache folder content

argopy.tutorial.open_dataset

open_dataset(*name: str*) → tuple

Open a dataset from the argopy online data repository (requires internet).

If a local copy is found then always use that to avoid network traffic.

Refresh dataset with: ` `argopy.tutorial.repodata().download(overwrite=True)` `

Parameters

name (*str*) – Name of the dataset to load or get information for. It can be one of the following:

- `gdac`: A small subset of the GDAC ftp.
- `weekly_index_prof`: The weekly profile index file
- `global_index_prof`: The global profile index file

Returns

- **path** (*str*) – Root path to files
- **files** (*list(str) or str*) – List of files with the requested dataset

argopy.show_versions

show_versions(*file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>, conda=False*)

Print the versions of argopy and its dependencies

Parameters

- **file** (*file-like, optional*) – print to the given file-like object. Defaults to sys.stdout.
- **conda** (*bool, optional*) – format versions to be copy/pasted on a conda environment file (default, False)

argopy.xarray.ArgoEngine

class ArgoEngine

Backend for Argo netCDF files based on the xarray netCDF4 engine

It can open any Argo ".nc" files with 'Argo' in their global attribute 'Conventions'.

But it will not be detected as valid backend for netcdf files, so make sure to specify `engine="argo"` in `xarray.open_dataset()`.

Examples

```
>>> import xarray as xr
>>> ds = xr.open_dataset("dac/aoml/1901393/1901393_prof.nc", engine='argo')
```

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
guess_can_open(filename_or_obj)	Backend open_dataset method used by Xarray in `open_dataset()`.
open_dataset(filename_or_obj, *[, ...])	Backend open_dataset method used by Xarray in `open_dataset()`.

Attributes

available
description
open_dataset_parameters
url

1.12.7 Internals

File systems

argopy.stores.filestore([cache, cachedir])	Argo local file system
argopy.stores.httpstore([cache, cachedir])	Argo http file system
argopy.stores.memorystore([cache, cachedir])	Argo in-memory file system (global)
argopy.stores.ftpstore([cache, cachedir])	Argo ftp file system

argopy.stores.filestore

class filestore(*cache: bool = False*, *cachedir: str = ''*, ***kwargs*)

Argo local file system

Relies on `fsspec.implementations.local.LocalFileSystem`

__init__(*cache: bool = False*, *cachedir: str = ''*, ***kwargs*)

Create a file storage system for Argo data

Parameters

- **cache** (*bool (False)*) –
- **cachedir** (*str (from OPTIONS)*) –
- ****kwargs** (*(optional)*) – Other arguments passed to `fsspec.filesystem`

Methods

__init__([cache, cachedir])	Create a file storage system for Argo data
cachepath(uri[, errors])	Return path to cached file for a given URI
clear_cache()	Remove cache files and entry from uri open with this store instance
exists(path, *args)	
expand_path(path)	
glob(path, **kwargs)	
open(path, *args, **kwargs)	
open_dataset(path, *args, **kwargs)	Return a xarray.dataset from a path.
open_json(url, **kwargs)	Return a json from a path, or verbose errors
open_mfdataset(urls[, concat_dim, ...])	Open multiple urls as a single xarray dataset.
read_csv(path, **kwargs)	Return a pandas.dataframe from a path that is a csv resource
register(uri)	Keep track of files open with this instance
store_path(uri)	

Attributes

cached_files	
protocol	File system name, one in fsspec.registry.known_implementations

argopy.stores.httpstore

class httpstore(*cache: bool = False*, *cachedir: str = ''*, ***kwargs*)

Argo http file system

Relies on `fsspec.implementations.http.HTTPFileSystem`

This store intends to make argopy: safer to failures from http requests and to provide higher levels methods to work with our datasets

This store is primarily used by the Erddap/Argovis data/index fetchers

__init__(*cache: bool = False*, *cachedir: str = ''*, ***kwargs*)

Create a file storage system for Argo data

Parameters

- **cache** (*bool (False)*) –
- **cachedir** (*str (from OPTIONS)*) –
- ****kwargs** (*(optional)*) – Other arguments passed to `fsspec.filesystem`

Methods

__init__([cache, cachedir])	Create a file storage system for Argo data
cachepath(uri[, errors])	Return path to cached file for a given URI
clear_cache()	Remove cache files and entry from uri open with this store instance
curateurl(url)	Possibly replace server of a given url by a local argopy option value
download_url(url[, n_attempt, max_attempt, ...])	URL data downloader
exists(path, *args)	
expand_path(path)	
glob(path, **kwargs)	
open(path, *args, **kwargs)	
open_dataset(url, **kwargs)	Open and decode a xarray dataset from an url
open_json(url, **kwargs)	Return a json from an url, or verbose errors
open_mfdataset(urls[, max_workers, method, ...])	Open multiple urls as a single xarray dataset.
open_mfjson(urls[, max_workers, method, ...])	Open multiple json urls
read_csv(url, **kwargs)	Read a comma-separated values (csv) url into Pandas DataFrame.
register(uri)	Keep track of files open with this instance
store_path(uri)	

Attributes

cached_files	
protocol	File system name, one in fsspec.registry.known_implementations

argopy.stores.memorystore

class memorystore(*cache: bool = False*, *cachedir: str = ''*, ***kwargs*)

Argo in-memory file system (global)

Note that this inherits from *argopy.stores.filestore*, not the:class:*argopy.stores.argo_store_proto*.

Relies on fsspec.implementations.memory.MemoryFileSystem

__init__(*cache: bool = False*, *cachedir: str = ''*, ***kwargs*)

Create a file storage system for Argo data

Parameters

- **cache** (*bool (False)*) –
- **cachedir** (*str (from OPTIONS)*) –
- ****kwargs** (*(optional)*) – Other arguments passed to fsspec.filesystem

Methods

__init__([cache, cachedir])	Create a file storage system for Argo data
cachepath(uri[, errors])	Return path to cached file for a given URI
clear_cache()	Remove cache files and entry from uri open with this store instance
exists(path, *args)	Check if path can be open or not
expand_path(path)	
glob(path, **kwargs)	
open(path, *args, **kwargs)	
open_dataset(path, *args, **kwargs)	Return a xarray.dataset from a path.
open_json(url, **kwargs)	Return a json from a path, or verbose errors
open_mfdataset(urls[, concat_dim, ...])	Open multiple urls as a single xarray dataset.
read_csv(path, **kwargs)	Return a pandas.dataframe from a path that is a csv resource
register(uri)	Keep track of files open with this instance
store_path(uri)	

Attributes

cached_files	
protocol	File system name, one in fsspec.registry.known_implementations

argopy.stores.ftpstore

class ftpstore(*cache: bool = False*, *cachedir: str = ''*, ***kwargs*)

Argo ftp file system

Relies on `fsspec.implementations.ftp.FTPFileSystem`

__init__(*cache: bool = False*, *cachedir: str = ''*, ***kwargs*)

Create a file storage system for Argo data

Parameters

- **cache** (*bool (False)*) –
- **cachedir** (*str (from OPTIONS)*) –
- ****kwargs** (*(optional)*) – Other arguments passed to `fsspec.filesystem`

Methods

__init__([cache, cachedir])	Create a file storage system for Argo data
cachepath(uri[, errors])	Return path to cached file for a given URI
clear_cache()	Remove cache files and entry from uri open with this store instance
curateurl(url)	Possibly replace server of a given url by a local argopy option value
download_url(url[, n_attempt, max_attempt, ...])	URL data downloader
exists(path, *args)	
expand_path(path)	
glob(path, **kwargs)	
open(path, *args, **kwargs)	
open_dataset(url, *args, **kwargs)	Open and decode a xarray dataset from an ftp url
open_json(url, **kwargs)	Return a json from an url, or verbose errors
open_mfdataset(urls[, max_workers, method, ...])	Open multiple ftp urls as a single xarray dataset.
open_mfjson(urls[, max_workers, method, ...])	Open multiple json urls
read_csv(url, **kwargs)	Read a comma-separated values (csv) url into Pandas DataFrame.
register(uri)	Keep track of files open with this instance
store_path(uri)	

Attributes

cached_files	
protocol	File system name, one in fsspec.registry.known_implementations

Argo index store

ArgoIndex([host, index_file, convention, ...])	Argo GDAC index store
argopy.stores.indexstore_pa	alias of indexstore_pyarrow
argopy.stores.indexstore_pd	alias of indexstore_pandas

argopy.stores.indexstore_pa

indexstore_pa
alias of indexstore_pyarrow

argopy.stores.indexstore_pd

indexstore_pd
alias of indexstore_pandas

Fetcher sources

ERDDAP

argopy.data_fetchers.erddap_data.ErddapArgoDataFetcher([...])	Manage access to Argo data through Ifremer ERDDAP
argopy.data_fetchers.erddap_data.Fetch_wmo([...])	Manage access to Argo data through Ifremer ERDDAP for: a list of WMOs
argopy.data_fetchers.erddap_data.Fetch_box([...])	Manage access to Argo data through Ifremer ERDDAP for: an ocean rectangle

argopy.data_fetchers.erddap_data.ErddapArgoDataFetcher

class ErddapArgoDataFetcher(*ds: str = ''*, *cache: bool = False*, *cachedir: str = ''*, *parallel: bool = False*, *parallel_method: str = 'erddap'*, *progress: bool = False*, *chunks: str = 'auto'*, *chunks_maxsize: dict = {}*, *api_timeout: int = 0*, *params: str | list = 'all'*, *measured: str | list | None = None*, ***kwargs*)

Manage access to Argo data through Ifremer ERDDAP

ERDDAP transaction are managed with the erddapy library

This class is a prototype not meant to be instantiated directly

__init__(*ds: str = '', cache: bool = False, cachedir: str = '', parallel: bool = False, parallel_method: str = 'erddap', progress: bool = False, chunks: str = 'auto', chunks_maxsize: dict = {}, api_timeout: int = 0, params: str | list = 'all', measured: str | list | None = None, **kwargs*)

Instantiate an ERDDAP Argo data fetcher

Parameters

- **ds** (*str (optional)*) – Dataset to load: 'phy' or 'ref' or 'bgc'
- **cache** (*bool (optional)*) – Cache data or not (default: False)
- **cachedir** (*str (optional)*) – Path to cache folder
- **parallel** (*bool (optional)*) – Chunk request to use parallel fetching (default: False)
- **parallel_method** (*str (optional)*) – Define the parallelization method: `thread`, `process` or a `dask.distributed.client.Client`.
- **progress** (*bool (optional)*) – Show a progress bar or not when `parallel` is set to True.
- **chunks** (*'auto' or dict of integers (optional)*) – Dictionary with request access point as keys and number of chunks to create as values. Eg: {'wmo': 10} will create a maximum of 10 chunks along WMOs when used with `Fetch_wmo`.
- **chunks_maxsize** (*dict (optional)*) – Dictionary with request access point as keys and chunk size as values (used as maximum values in 'auto' chunking). Eg: {'wmo': 5} will create chunks with as many as 5 WMOs each.
- **api_timeout** (*int (optional)*) – Erddap request time out in seconds. Set to OPTIONS['api_timeout'] by default.
- **params** (*Union[str, list] (optional, default='all')*) – List of BGC essential variables to retrieve, i.e. that will be in the output `xr.DataSet`. By default, this is set to `all`, i.e. any variable found in at least of the profile in the data selection will be included in the output.
- **measured** (*Union[str, list] (optional, default=None)*) – List of BGC essential variables that can't be NaN. If set to 'all', this is an easy way to reduce the size of the `xr.DataSet` to points where all variables have been measured. Otherwise, provide a simple list of variables.

Methods

`__init__`([ds, cache, cachedir, parallel, ...])	Instantiate an ERDDAP Argo data fetcher
`clear_cache`()	Remove cache files and entries from resources opened with this fetcher
`cname`()	Return a unique string defining the constraints
`dashboard`(**kw)	Return 3rd party dashboard for the access point
`define_constraints`()	Define erddapy constraints
`filter_data_mode`(ds, **kwargs)	Apply xarray argo accessor filter_data_mode method
`filter_points`(ds)	Enforce request criteria
`filter_qc`(ds, **kwargs)	Apply xarray argo accessor filter_qc method
`filter_researchmode`(ds, *args, **kwargs)	Filter dataset for research user mode
`filter_variables`(ds, mode, *args, **kwargs)	Filter variables according to user mode
`get_url`()	Return the URL to download requested data
`init`(*args, **kwargs)	Initialisation for a specific fetcher
`post_process`(this_ds[, add_dm, URI])	Post-process a xarray.DataSet created from a netcdf erddap response
`to_xarray`([errors, add_dm, concat, max_workers])	Load Argo data and return a xarray.DataSet

Attributes

`N_POINTS`	Number of measurements expected to be returned by a request
`cachepath`	Return path to cached file(s) for this request
`server`	URL of the Erddap server
`sha`	Returns a unique SHA for a specific cname / fetcher implementation
`uri`	Return the list of Unique Resource Identifier (URI) to download data

argopy.data_fetchers.erddap_data.Fetch_wmo

class Fetch_wmo(*ds: str = ''*, *cache: bool = False*, *cachedir: str = ''*, *parallel: bool = False*, *parallel_method: str = 'erddap'*, *progress: bool = False*, *chunks: str = 'auto'*, *chunks_maxsize: dict = {}*, *api_timeout: int = 0*, *params: str | list = 'all'*, *measured: str | list | None = None*, ***kwargs*)

Manage access to Argo data through Ifremer ERDDAP for: a list of WMOs

This class is instantiated when a call is made to these facade access points:

- *ArgoDataFetcher(src='erddap').float(**)*
- *ArgoDataFetcher(src='erddap').profile(**)*

__init__(*ds: str = ''*, *cache: bool = False*, *cachedir: str = ''*, *parallel: bool = False*, *parallel_method: str = 'erddap'*, *progress: bool = False*, *chunks: str = 'auto'*, *chunks_maxsize: dict = {}*, *api_timeout: int = 0*, *params: str | list = 'all'*, *measured: str | list | None = None*, ***kwargs*)

Instantiate an ERDDAP Argo data fetcher

Parameters

- **ds** (*str (optional)*) – Dataset to load: 'phy' or 'ref' or 'bgc'

- **cache** (*bool (optional)*) – Cache data or not (default: False)
- **cachedir** (*str (optional)*) – Path to cache folder
- **parallel** (*bool (optional)*) – Chunk request to use parallel fetching (default: False)
- **parallel_method** (*str (optional)*) – Define the parallelization method: `thread`, `process` or a `dask.distributed.client.Client`.
- **progress** (*bool (optional)*) – Show a progress bar or not when `parallel` is set to True.
- **chunks** (*'auto' or dict of integers (optional)*) – Dictionary with request access point as keys and number of chunks to create as values. Eg: {'wmo': 10} will create a maximum of 10 chunks along WMOs when used with `Fetch_wmo`.
- **chunks_maxsize** (*dict (optional)*) – Dictionary with request access point as keys and chunk size as values (used as maximum values in 'auto' chunking). Eg: {'wmo': 5} will create chunks with as many as 5 WMOs each.
- **api_timeout** (*int (optional)*) – Erddap request time out in seconds. Set to OPTIONS['api_timeout'] by default.
- **params** (*Union[str, list] (optional, default='all')*) – List of BGC essential variables to retrieve, i.e. that will be in the output `xr.DataSet`. By default, this is set to `all`, i.e. any variable found in at least of the profile in the data selection will be included in the output.
- **measured** (*Union[str, list] (optional, default=None)*) – List of BGC essential variables that can't be NaN. If set to 'all', this is an easy way to reduce the size of the `xr.DataSet`` to points where all variables have been measured. Otherwise, provide a simple list of variables.

Methods

__init__([ds, cache, cachedir, parallel, ...])	Instantiate an ERDDAP Argo data fetcher
clear_cache()	Remove cache files and entries from resources opened with this fetcher
cname()	Return a unique string defining the constraints
dashboard(**kw)	Return 3rd party dashboard for the access point
define_constraints()	Define erddap constraints
filter_data_mode(ds, **kwargs)	Apply xarray argo accessor filter_data_mode method
filter_points(ds)	Enforce request criteria
filter_qc(ds, **kwargs)	Apply xarray argo accessor filter_qc method
filter_researchmode(ds, *args, **kwargs)	Filter dataset for research user mode
filter_variables(ds, mode, *args, **kwargs)	Filter variables according to user mode
get_url()	Return the URL to download requested data
init([WMO, CYC])	Create Argo data loader for WMOs
post_process(this_ds[, add_dm, URI])	Post-process a xarray.DataSet created from a netcdf erddap response
to_xarray([errors, add_dm, concat, max_workers])	Load Argo data and return a xarray.DataSet

Attributes

N_POINTS	Number of measurements expected to be returned by a request
cachepath	Return path to cached file(s) for this request
server	URL of the Erddap server
sha	Returns a unique SHA for a specific cname / fetcher implementation
uri	List of URLs to load for a request

argopy.data_fetchers.erddap_data.Fetch_box

class Fetch_box(*ds: str = ''*, *cache: bool = False*, *cachedir: str = ''*, *parallel: bool = False*, *parallel_method: str = 'erddap'*, *progress: bool = False*, *chunks: str = 'auto'*, *chunks_maxsize: dict = {}*, *api_timeout: int = 0*, *params: str | list = 'all'*, *measured: str | list | None = None*, ***kwargs*)

Manage access to Argo data through Ifremer ERDDAP for: an ocean rectangle

__init__(*ds: str = ''*, *cache: bool = False*, *cachedir: str = ''*, *parallel: bool = False*, *parallel_method: str = 'erddap'*, *progress: bool = False*, *chunks: str = 'auto'*, *chunks_maxsize: dict = {}*, *api_timeout: int = 0*, *params: str | list = 'all'*, *measured: str | list | None = None*, ***kwargs*)

Instantiate an ERDDAP Argo data fetcher

Parameters

- **ds** (*str (optional)*) – Dataset to load: 'phy' or 'ref' or 'bgc'
- **cache** (*bool (optional)*) – Cache data or not (default: False)
- **cachedir** (*str (optional)*) – Path to cache folder
- **parallel** (*bool (optional)*) – Chunk request to use parallel fetching (default: False)
- **parallel_method** (*str (optional)*) – Define the parallelization method: `thread`, `process` or a `dask.distributed.client.Client`.
- **progress** (*bool (optional)*) – Show a progress bar or not when `parallel` is set to True.
- **chunks** (*'auto' or dict of integers (optional)*) – Dictionary with request access point as keys and number of chunks to create as values. Eg: {'wmo': 10} will create a maximum of 10 chunks along WMOs when used with `Fetch_wmo`.
- **chunks_maxsize** (*dict (optional)*) – Dictionary with request access point as keys and chunk size as values (used as maximum values in 'auto' chunking). Eg: {'wmo': 5} will create chunks with as many as 5 WMOs each.
- **api_timeout** (*int (optional)*) – Erddap request time out in seconds. Set to OPTIONS['api_timeout'] by default.
- **params** (*Union[str, list] (optional, default='all')*) – List of BGC essential variables to retrieve, i.e. that will be in the output `xr.DataSet`. By default, this is set to `all`, i.e. any variable found in at least of the profile in the data selection will be included in the output.
- **measured** (*Union[str, list] (optional, default=None)*) – List of BGC essential variables that can't be NaN. If set to 'all', this is an easy way to reduce the size of

the `xr.DataSet` to points where all variables have been measured. Otherwise, provide a simple list of variables.

Methods

__init__([ds, cache, cachedir, parallel, ...])	Instantiate an ERDDAP Argo data fetcher
clear_cache()	Remove cache files and entries from resources opened with this fetcher
cname()	Return a unique string defining the constraints
dashboard(**kw)	Return 3rd party dashboard for the access point
define_constraints()	Define request constraints
filter_data_mode(ds, **kwargs)	Apply xarray argo accessor filter_data_mode method
filter_points(ds)	Enforce request criteria
filter_qc(ds, **kwargs)	Apply xarray argo accessor filter_qc method
filter_researchmode(ds, *args, **kwargs)	Filter dataset for research user mode
filter_variables(ds, mode, *args, **kwargs)	Filter variables according to user mode
get_url()	Return the URL to download requested data
init(box, **kw)	Create Argo data loader
post_process(this_ds[, add_dm, URI])	Post-process a xarray.DataSet created from a netcdf erddap response
to_xarray([errors, add_dm, concat, max_workers])	Load Argo data and return a xarray.DataSet

Attributes

N_POINTS	Number of measurements expected to be returned by a request
cachepath	Return path to cached file(s) for this request
server	URL of the Erddap server
sha	Returns a unique SHA for a specific cname / fetcher implementation
uri	List of files to load for a request

GDAC

argopy.data_fetchers.gdacftp_data.FTPArgoDataFetcher([...])	Manage access to Argo data from a remote GDAC FTP.
argopy.data_fetchers.gdacftp_data.Fetch_wmo([...])	Manage access to GDAC ftp Argo data for: a list of WMOs.
argopy.data_fetchers.gdacftp_data.Fetch_box([...])	Manage access to GDAC ftp Argo data for: a rectangular space/time domain.

argopy.data_fetchers.gdacftp_data.FTPArgoDataFetcher

class **FTPArgoDataFetcher**(*ftp: str = '', ds: str = '', cache: bool = False, cachedir: str = '', dimension: str = 'point', errors: str = 'raise', parallel: bool = False, parallel_method: str = 'thread', progress: bool = False, api_timeout: int = 0, **kwargs*)

Manage access to Argo data from a remote GDAC FTP.

> **Warning:** This class is a prototype not meant to be instantiated directly

__init__(*ftp: str = '', ds: str = '', cache: bool = False, cachedir: str = '', dimension: str = 'point', errors: str = 'raise', parallel: bool = False, parallel_method: str = 'thread', progress: bool = False, api_timeout: int = 0, **kwargs*)

Init fetcher

Parameters

- **ftp** (*str (optional)*) – Path to the remote FTP directory where the 'dac' folder is located.
- **ds** (*str (optional)*) – Dataset to load: 'phy' or 'bgc'
- **cache** (*bool (optional)*) – Cache data or not (default: False)
- **cachedir** (*str (optional)*) – Path to cache folder
- **dimension** (*str, default: 'point'*) – Main dimension of the output dataset. This can be "profile" to retrieve a collection of profiles, or "point" (default) to have data as a collection of measurements. This can be used to optimise performances.
- **errors** (*str (optional)*) – If set to 'raise' (default), will raise a NetCDF4FileNotFoundError error if any of the requested files cannot be found. If set to 'ignore', the file not found is skipped when fetching data.
- **parallel** (*bool (optional)*) – Chunk request to use parallel fetching (default: False)
- **parallel_method** (*str (optional)*) – Define the parallelization method: `thread`, `process` or a `dask.distributed.client.Client`.
- **progress** (*bool (optional)*) – Show a progress bar or not when fetching data.
- **api_timeout** (*int (optional)*) – FTP request time out in seconds. Set to OPTIONS['api_timeout'] by default.

Methods

`__init__`([ftp, ds, cache, cachedir, ...])	Init fetcher
`clear_cache`()	Remove cached files and entries from resources opened with this fetcher
`cname`()	Return a unique string defining the constraints
`dashboard`(**kw)	Return 3rd party dashboard for the access point
`filter_data_mode`(ds, **kwargs)	
`filter_points`(ds)	Enforce request criteria
`filter_qc`(ds, **kwargs)	
`filter_researchmode`(ds, *args, **kwargs)	Filter dataset for research user mode
`filter_variables`(ds, mode, *args, **kwargs)	Filter variables according to user mode
`init`(*args, **kwargs)	Initialisation for a specific fetcher
`to_xarray`([errors])	Load Argo data and return a `xarray.Dataset`
`uri_mono2multi`(URIs)	Convert mono-profile URI files to multi-profile files

Attributes

`cachepath`	Return path to cache file(s) for this request
`sha`	Returns a unique SHA for a specific cname / fetcher implementation
`uri`	Return the list of files to load

argopy.data_fetchers.gdacftp_data.Fetch_wmo

class Fetch_wmo(*ftp: str = '', ds: str = '', cache: bool = False, cachedir: str = '', dimension: str = 'point', errors: str = 'raise', parallel: bool = False, parallel_method: str = 'thread', progress: bool = False, api_timeout: int = 0, **kwargs*)

Manage access to GDAC ftp Argo data for: a list of WMOs.

This class is instantiated when a call is made to these facade access points:

```
>>> ArgoDataFetcher(src='gdac').float(**)
>>> ArgoDataFetcher(src='gdac').profile(**)
```

__init__(*ftp: str = '', ds: str = '', cache: bool = False, cachedir: str = '', dimension: str = 'point', errors: str = 'raise', parallel: bool = False, parallel_method: str = 'thread', progress: bool = False, api_timeout: int = 0, **kwargs*)

Init fetcher

Parameters

- **ftp** (*str (optional)*) – Path to the remote FTP directory where the 'dac' folder is located.
- **ds** (*str (optional)*) – Dataset to load: 'phy' or 'bgc'
- **cache** (*bool (optional)*) – Cache data or not (default: False)
- **cachedir** (*str (optional)*) – Path to cache folder

- **dimension** (*str, default: 'point'*) – Main dimension of the output dataset. This can be "profile" to retrieve a collection of profiles, or "point" (default) to have data as a collection of measurements. This can be used to optimise performances.
- **errors** (*str (optional)*) – If set to 'raise' (default), will raise a NetCDF4FileNotFoundError error if any of the requested files cannot be found. If set to 'ignore', the file not found is skipped when fetching data.
- **parallel** (*bool (optional)*) – Chunk request to use parallel fetching (default: False)
- **parallel_method** (*str (optional)*) – Define the parallelization method: `thread`, `process` or a `dask.distributed.client.Client`.
- **progress** (*bool (optional)*) – Show a progress bar or not when fetching data.
- **api_timeout** (*int (optional)*) – FTP request time out in seconds. Set to OPTIONS['api_timeout'] by default.

Methods

__init__([ftp, ds, cache, cachedir, ...])	Init fetcher
clear_cache()	Remove cached files and entries from resources opened with this fetcher
cname()	Return a unique string defining the constraints
dashboard(**kw)	Return 3rd party dashboard for the access point
filter_data_mode(ds, **kwargs)	
filter_points(ds)	Enforce request criteria
filter_qc(ds, **kwargs)	
filter_researchmode(ds, *args, **kwargs)	Filter dataset for research user mode
filter_variables(ds, mode, *args, **kwargs)	Filter variables according to user mode
init([WMO, CYC])	Create Argo data loader for WMOs
to_xarray([errors])	Load Argo data and return a `xarray.Dataset`
uri_mono2multi(URIs)	Convert mono-profile URI files to multi-profile files

Attributes

cachepath	Return path to cache file(s) for this request
sha	Returns a unique SHA for a specific cname / fetcher implementation
uri	List of files to load for a request

argopy.data_fetchers.gdacftp_data.Fetch_box

class Fetch_box(*ftp: str = '', ds: str = '', cache: bool = False, cachedir: str = '', dimension: str = 'point', errors: str = 'raise', parallel: bool = False, parallel_method: str = 'thread', progress: bool = False, api_timeout: int = 0, **kwargs*)

Manage access to GDAC ftp Argo data for: a rectangular space/time domain.

This class is instantiated when a call is made to these facade access points:

```
>>> ArgoDataFetcher(src='gdac').region(**)
```

__init__(*ftp: str = '', ds: str = '', cache: bool = False, cachedir: str = '', dimension: str = 'point', errors: str = 'raise', parallel: bool = False, parallel_method: str = 'thread', progress: bool = False, api_timeout: int = 0, **kwargs*)

Init fetcher

Parameters

- **ftp** (*str (optional)*) – Path to the remote FTP directory where the 'dac' folder is located.
- **ds** (*str (optional)*) – Dataset to load: 'phy' or 'bgc'
- **cache** (*bool (optional)*) – Cache data or not (default: False)
- **cachedir** (*str (optional)*) – Path to cache folder
- **dimension** (*str, default: 'point'*) – Main dimension of the output dataset. This can be "profile" to retrieve a collection of profiles, or "point" (default) to have data as a collection of measurements. This can be used to optimise performances.
- **errors** (*str (optional)*) – If set to 'raise' (default), will raise a NetCDF4FileNotFoundError error if any of the requested files cannot be found. If set to 'ignore', the file not found is skipped when fetching data.
- **parallel** (*bool (optional)*) – Chunk request to use parallel fetching (default: False)
- **parallel_method** (*str (optional)*) – Define the parallelization method: `thread`, `process` or a `dask.distributed.client.Client`.
- **progress** (*bool (optional)*) – Show a progress bar or not when fetching data.
- **api_timeout** (*int (optional)*) – FTP request time out in seconds. Set to OPTIONS['api_timeout'] by default.

Methods

`__init__`([ftp, ds, cache, cachedir, ...])	Init fetcher
`clear_cache`()	Remove cached files and entries from resources opened with this fetcher
`cname`()	Return a unique string defining the constraints
`dashboard`(**kw)	Return 3rd party dashboard for the access point
`filter_data_mode`(ds, **kwargs)	
`filter_points`(ds)	Enforce request criteria
`filter_qc`(ds, **kwargs)	
`filter_researchmode`(ds, *args, **kwargs)	Filter dataset for research user mode
`filter_variables`(ds, mode, *args, **kwargs)	Filter variables according to user mode
`init`(box[, nrows])	Create Argo data loader
`to_xarray`([errors])	Load Argo data and return a `xarray.Dataset`
`uri_mono2multi`(URIs)	Convert mono-profile URI files to multi-profile files

Attributes

`cachepath`	Return path to cache file(s) for this request
`sha`	Returns a unique SHA for a specific cname / fetcher implementation
`uri`	List of files to load for a request

Argovis

argopy.data_fetchers.argovis_data.ArgovisDataFetcher([...])	
argopy.data_fetchers.argovis_data.Fetch_wmo([...])	
argopy.data_fetchers.argovis_data.Fetch_box([...])	

argopy.data_fetchers.argovis_data.ArgovisDataFetcher

class ArgovisDataFetcher(*ds: str = ''*, *cache: bool = False*, *cachedir: str = ''*, *parallel: bool = False*, *parallel_method: str = 'thread'*, *progress: bool = False*, *chunks: str = 'auto'*, *chunks_maxsize: dict = {}*, *api_timeout: int = 0*, ***kwargs*)

__init__(*ds: str = ''*, *cache: bool = False*, *cachedir: str = ''*, *parallel: bool = False*, *parallel_method: str = 'thread'*, *progress: bool = False*, *chunks: str = 'auto'*, *chunks_maxsize: dict = {}*, *api_timeout: int = 0*, ***kwargs*)

Instantiate an Argovis Argo data loader

Parameters

- **ds** (*str (optional)*) – Dataset to load: 'phy' or 'bgc'

- **cache** (*bool (optional)*) – Cache data or not (default: False)
- **cachedir** (*str (optional)*) – Path to cache folder
- **parallel** (*bool (optional)*) – Chunk request to use parallel fetching (default: False)
- **parallel_method** (*str (optional)*) – Define the parallelization method: `thread`, `process` or a `dask.distributed.client.Client`.
- **progress** (*bool (optional)*) – Show a progress bar or not when `parallel` is set to True.
- **chunks** (*'auto' or dict of integers (optional)*) – Dictionary with request access point as keys and number of chunks to create as values. Eg: {'wmo': 10} will create a maximum of 10 chunks along WMOs when used with `Fetch_wmo`.
- **chunks_maxsize** (*dict (optional)*) – Dictionary with request access point as keys and chunk size as values (used as maximum values in 'auto' chunking). Eg: {'wmo': 5} will create chunks with as many as 5 WMOs each.
- **api_timeout** (*int (optional)*) – Argovis API request time out in seconds. Set to OPTIONS['api_timeout'] by default.

Methods

__init__([ds, cache, cachedir, parallel, ...])	Instantiate an Argovis Argo data loader
clear_cache()	Remove cache files and entries from resources opened with this fetcher
cname()	Return a unique string defining the constraints
dashboard(**kw)	Return 3rd party dashboard for the access point
filter_data_mode(ds, **kwargs)	
filter_domain(ds)	Enforce rectangular box shape
filter_qc(ds, **kwargs)	
filter_researchmode(ds, *args, **kwargs)	Filter dataset for research user mode
filter_variables(ds, mode, *args, **kwargs)	Filter variables according to user mode
init(*args, **kwargs)	Initialisation for a specific fetcher
json2dataframe(profiles)	convert json data to Pandas DataFrame
to_dataframe([errors])	Load Argo data and return a Pandas dataframe
to_xarray([errors])	Download and return data as xarray Datasets
url_encode(urls)	Return safely encoded list of urls

Attributes

cachepath	Return path to cache file for this request
sha	Returns a unique SHA for a specific cname / fetcher implementation
uri	Return the URL used to download data

argopy.data_fetchers.argovis_data.Fetch_wmo

class Fetch_wmo(*ds: str = ''*, *cache: bool = False*, *cachedir: str = ''*, *parallel: bool = False*, *parallel_method: str = 'thread'*, *progress: bool = False*, *chunks: str = 'auto'*, *chunks_maxsize: dict = {}*, *api_timeout: int = 0*, ***kwargs*)

__init__(*ds: str = ''*, *cache: bool = False*, *cachedir: str = ''*, *parallel: bool = False*, *parallel_method: str = 'thread'*, *progress: bool = False*, *chunks: str = 'auto'*, *chunks_maxsize: dict = {}*, *api_timeout: int = 0*, ***kwargs*)

Instantiate an Argovis Argo data loader

Parameters

- **ds** (*str (optional)*) – Dataset to load: 'phy' or 'bgc'
- **cache** (*bool (optional)*) – Cache data or not (default: False)
- **cachedir** (*str (optional)*) – Path to cache folder
- **parallel** (*bool (optional)*) – Chunk request to use parallel fetching (default: False)
- **parallel_method** (*str (optional)*) – Define the parallelization method: `thread`, `process` or a `dask.distributed.client.Client`.
- **progress** (*bool (optional)*) – Show a progress bar or not when `parallel` is set to True.
- **chunks** (*'auto' or dict of integers (optional)*) – Dictionary with request access point as keys and number of chunks to create as values. Eg: {'wmo': 10} will create a maximum of 10 chunks along WMOs when used with `Fetch_wmo`.
- **chunks_maxsize** (*dict (optional)*) – Dictionary with request access point as keys and chunk size as values (used as maximum values in 'auto' chunking). Eg: {'wmo': 5} will create chunks with as many as 5 WMOs each.
- **api_timeout** (*int (optional)*) – Argovis API request time out in seconds. Set to OPTIONS['api_timeout'] by default.

Methods

`__init__`([ds, cache, cachedir, parallel, ...])	Instantiate an Argovis Argo data loader
`clear_cache`()	Remove cache files and entries from resources opened with this fetcher
`cname`()	Return a unique string defining the constraints
`dashboard`(**kw)	Return 3rd party dashboard for the access point
`filter_data_mode`(ds, **kwargs)	
`filter_domain`(ds)	Enforce rectangular box shape
`filter_qc`(ds, **kwargs)	
`filter_researchmode`(ds, *args, **kwargs)	Filter dataset for research user mode
`filter_variables`(ds, mode, *args, **kwargs)	Filter variables according to user mode
`get_url`(wmo[, cyc])	Return path toward the source file of a given wmo/cyc pair
`init`([WMO, CYC])	Create Argo data loader for WMOs and CYCs
`json2dataframe`(profiles)	convert json data to Pandas DataFrame
`to_dataframe`([errors])	Load Argo data and return a Pandas dataframe
`to_xarray`([errors])	Download and return data as xarray Datasets
`url_encode`(urls)	Return safely encoded list of urls

Attributes

`cachepath`	Return path to cache file for this request
`sha`	Returns a unique SHA for a specific cname / fetcher implementation
`uri`	List of URLs to load for a request

argopy.data_fetchers.argovis_data.Fetch_box

class Fetch_box(*ds: str = ''*, *cache: bool = False*, *cachedir: str = ''*, *parallel: bool = False*, *parallel_method: str = 'thread'*, *progress: bool = False*, *chunks: str = 'auto'*, *chunks_maxsize: dict = {}*, *api_timeout: int = 0*, ***kwargs*)

__init__(*ds: str = ''*, *cache: bool = False*, *cachedir: str = ''*, *parallel: bool = False*, *parallel_method: str = 'thread'*, *progress: bool = False*, *chunks: str = 'auto'*, *chunks_maxsize: dict = {}*, *api_timeout: int = 0*, ***kwargs*)

Instantiate an Argovis Argo data loader

Parameters

- **ds** (*str (optional)*) – Dataset to load: 'phy' or 'bgc'
- **cache** (*bool (optional)*) – Cache data or not (default: False)
- **cachedir** (*str (optional)*) – Path to cache folder
- **parallel** (*bool (optional)*) – Chunk request to use parallel fetching (default: False)
- **parallel_method** (*str (optional)*) – Define the parallelization method: `thread`, `process` or a `dask.distributed.client.Client`.

- **progress** (*bool (optional)*) – Show a progress bar or not when `parallel` is set to True.
- **chunks** (*'auto' or dict of integers (optional)*) – Dictionary with request access point as keys and number of chunks to create as values. Eg: {'wmo': 10} will create a maximum of 10 chunks along WMOs when used with `Fetch_wmo`.
- **chunks_maxsize** (*dict (optional)*) – Dictionary with request access point as keys and chunk size as values (used as maximum values in 'auto' chunking). Eg: {'wmo': 5} will create chunks with as many as 5 WMOs each.
- **api_timeout** (*int (optional)*) – Argovis API request time out in seconds. Set to OPTIONS['api_timeout'] by default.

Methods

__init__([ds, cache, cachedir, parallel, ...])	Instantiate an Argovis Argo data loader
clear_cache()	Remove cache files and entries from resources opened with this fetcher
cname()	Return a unique string defining the constraints
dashboard(**kw)	Return 3rd party dashboard for the access point
filter_data_mode(ds, **kwargs)	
filter_domain(ds)	Enforce rectangular box shape
filter_qc(ds, **kwargs)	
filter_researchmode(ds, *args, **kwargs)	Filter dataset for research user mode
filter_variables(ds, mode, *args, **kwargs)	Filter variables according to user mode
get_url()	
get_url_rect()	Return the URL used to download data
get_url_shape()	Return the URL used to download data
init(box, **kwargs)	Create Argo data loader
json2dataframe(profiles)	convert json data to Pandas DataFrame
to_dataframe([errors])	Load Argo data and return a Pandas dataframe
to_xarray([errors])	Download and return data as xarray Datasets
url_encode(urls)	Return safely encoded list of urls

Attributes

cachepath	Return path to cache file for this request
sha	Returns a unique SHA for a specific cname / fetcher implementation
uri	List of URLs to load for a request
url	

BIBLIOGRAPHY

[ADMT] See all the ADMT documentation here: http://www.argodatamgt.org/Documentation

[OWC] See all the details about the OWC methodology in these references:

[Guinehut2008] Guinehut, S., Coatanoan, C., Dhomps, A., Le Traon, P., & Larnicol, G. (2009). On the Use of Satellite Altimeter Data in Argo Quality Control, Journal of Atmospheric and Oceanic Technology, 26(2), 395-402. 10.1175/2008JTECHO648.1

INDEX

Symbols

A

B

C

D

E

F