**argopy** is a python library that aims to ease *Argo* data access, manipulation and visualisation for standard users as well as Argo experts.
Getting Started
  • Installation
  • Usage
  • Why argopy ?
  • What is Argo ?

1.1 Installation

1.1.1 Required dependencies
  • Python 3.6
  • Xarray 0.14
  • Erddapy 0.5
  • Fsspec 0.7
  • Gsw 3.3

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

The argopy software is continuously tested with success under all OS (Linux, Mac and Windows) and with python versions 3.6, 3.7 and 3.8.

1.1.2 Optional dependencies

For full plotting functionality the following packages are required:
  • Matplotlib 3.0 (mandatory)
  • Cartopy 0.17 (for some methods only)
  • Seaborn 0.9.0 (for some methods only)
1.1.3 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 version:

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

1.2 Usage

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

```python
from argopy import DataFetcher as ArgoDataFetcher
ds = ArgoDataFetcher().region([-75, -45, 20, 30, 0, 100, '2011', '2012']).to_xarray()
```

In this example, we used a data fetcher 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`:

```python
print(ds)
```

<`xarray.Dataset`

**Dimensions:**

| N_POINTS | 29023 |

**Coordinates:**

| LATITUDE     | float64 | 24.54 24.54 24.54 ... 22.58 22.58 |
| LONGITUDE    | float64 | -45.14 -45.14 ... -54.11 -54.11 |
| TIME         | datetime64[ns] | 2011-01-01T11:49:19 ... |

**Data variables:**

- **CONFIG_MISSION_NUMBER**: int64 1 1 1 1 1 1 1 1 ... 2 2 2 2 2 2 2 2
- **CYCLE_NUMBER**: int64 23 23 23 23 23 23 ... 23 23 23 23 23
- **DATA_MODE**: <U1 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D'
- **DIRECTION**: <U1 'A' 'A' 'A' 'A' ... 'A' 'A' 'A' 'A'
- **PLATFORM_NUMBER**: int64 1901463 1901463 ... 6901052 6901052
- **POSITION_QC**: int64 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1
- **PRES**: float64 5.0 10.0 15.0 ... 70.7 80.2 90.7
- **PRES_QC**: int64 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1
- **PSAL**: float64 37.45 37.45 37.45 ... 37.19 37.16
- **PSAL_QC**: int64 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1
- **TEMP**: float64 24.08 24.08 24.09 ... 24.78 24.14
- **TEMP_QC**: int64 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://www.ifremer.fr/erddap
- **Fetched_by**: docs
- **Fetched_date**: 2020/11/17
- **Fetched_constraints**: [x=-75.00/-45.00; y=20.00/30.00; z=0.0/100.0; t=201...
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 `argopy.argo.point2profile()`:

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

```
<xarray.Dataset>
Dimensions:
   (N_LEVELS: 55, N_PROF: 1419)
Coordinates:
   * N_PROF (N_PROF) int64 42 664 298 5 527 ... 368 526 1722 1375
   * N_LEVELS (N_LEVELS) int64 0 1 2 3 4 5 6 ... 49 50 51 52 53 54
   LATITUDE (N_PROF) float64 24.54 25.04 21.48 ... 20.66 22.58
   LONGITUDE (N_PROF) float64 -45.14 -51.58 ... -53.52 -54.11
   TIME (N_PROF) datetime64[ns] 2011-01-01T11:49:19 ... 20...
Data variables:
   CONFIG_MISSION_NUMBER (N_PROF) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
   CYCLE_NUMBER (N_PROF) int64 23 10 135 23 119 ... 193 171 165 4 23
   DATA_MODE (N_PROF) <U1 'D' 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D'
   DIRECTION (N_PROF) <U1 'A' 'A' 'A' 'A' 'A' ... 'A' 'A' 'A' 'A'
   PLATFORM_NUMBER (N_PROF) int64 1901463 4901211 ... 6900935 6901052
   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 ... nan nan
   PRES_QC (N_PROF) int64 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1
   PSAL (N_PROF, N_LEVELS) float64 37.45 37.45 ... 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
   TEMP (N_PROF, N_LEVELS) float64 24.08 24.08 ... nan nan
   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://www.ifremer.fr/erddap
   Fetched_by: docs
   Fetched_date: 2020/11/17
   Fetched_constraints: [x=-75.00/-45.00; y=20.00/30.00; z=0.0/100.0; t=201...
   Fetched_uri: ['https://www.ifremer.fr/erddap/tabledap/ArgoFloats...'
   history: Variables filtered according to DATA_MODE; Variable...
```

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

```python
[5]: ds = ArgoDataFetcher().float(6902746).to_xarray()
```

or for a float profile using the cycle number:

```python
[6]: 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 developing 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 confortable with the *standard* user mode.

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

In *standard* mode, fetched data are automatically filtered to account for their quality (only good are retained) and level of processing by the data centers (whether they looked at the data briefly or not).

Selecting user mode is further explained in the dedicated documentation section: *User mode: standard vs expert.*

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:

User Guide

- Fetching Argo data
- Data sources
- Manipulating data
- User mode: standard vs expert
- Fetching Argo meta-data
- Data visualisation
- Performance
1.5 Fetching Argo data

To access Argo data, you need to use a data fetcher. You can import and instantiate the default argopy data fetcher like this:

```python
from argopy import DataFetcher as ArgoDataFetcher
argo_loader = ArgoDataFetcher()
argo_loader
```

```
<datafetcher> 'Not initialised'
Current backend: erddap
Available fetchers: profile, float, region
User mode: standard
```

Then, you can request data for a specific space/time domain, for a given float or for a given vertical profile.

1.5.1 For a space/time domain

Use the fetcher access point `argopy.DataFetcher.region()` to specify a domain and chain with the `argopy.DataFetcher.to_xarray()` to get the data returned as `xarray.Dataset`.

For instance, to retrieve data from 75W to 45W, 20N to 30N, 0db to 10db and from January to May 2011:

```python
ds = argo_loader.region([-75, -45, 20, 30, 0, 10, '2011-01-01', '2011-06']).to_xarray()
print(ds)
```

```
<xarray.Dataset>
Dimensions:      (N_POINTS: 998)
Coordinates:
  * N_POINTS   (N_POINTS) int64 0 1 2 3 4 5 ... 993 994 995 996 997
    LATITUDE   (N_POINTS) float64 24.54 24.54 25.04 ... 24.96 24.96
    LONGITUDE  (N_POINTS) float64 -45.14 -45.14 ... -50.4 -50.4
    TIME       (N_POINTS) datetime64[ns] 2011-01-01T11:49:19 ... ...
Data variables:
  CONFIG_MISSION_NUMBER (N_POINTS) int64 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1
  CYCLE_NUMBER          (N_POINTS) int64 23 23 10 10 10 10 ... 2 10 10 38 38
  DATA_MODE             (N_POINTS) <U1 'D' 'D' 'D' 'D' ... 'D' 'D' 'D'
  DIRECTION             (N_POINTS) <U1 '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
  PRES                  (N_POINTS) float64 5.0 10.0 2.0 4.0 ... 9.42 5.0 10.0
  PRES_QC               (N_POINTS) int64 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1
  PSAL                  (N_POINTS) float64 37.45 37.45 37.28 ... 37.08 37.05
  PSAL_QC               (N_POINTS) int64 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 ... 25.1 24.79
  TEMP_QC               (N_POINTS) int64 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
Attributes:
  DATA_ID: ARGO
  DOI: http://doi.org/10.17882/42182
  Fetched_from: https://www.ifremer.fr/erddap
  Fetched_by: docs
  Fetched_date: 2020/11/17
  Fetched_constraints: [x=-75.00/-45.00; y=20.00/30.00; z=0.0/10.0; t=2011...
  Fetched_uri: ['https://www.ifremer.fr/erddap/tabledap/ArgoFloats...'
  history: Variables filtered according to DATA_MODE; Variable...
```
Note that:

- the constraints 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.

### 1.5.2 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 `argopy.DataFetcher.float()` to specify the float WMO platform number and chain with the `argopy.DataFetcher.to_xarray()` to get the data returned as `xarray.Dataset`.

For instance, to retrieve data for float WMO 6902746:

```python
[4]: ds = argo_loader.float(6902746).to_xarray()
print(ds)
```

```
<xarray.Dataset>
Dimensions: (N_POINTS: 9039)
Coordinates:
  * N_POINTS     (N_POINTS) int64 0 1 2 3 4 ... 9035 9036 9037 9038
  LATITUDE      (N_POINTS) float64 20.08 20.08 20.08 ... 16.3 16.3
  LONGITUDE    (N_POINTS) float64 -60.17 -60.17 ... -62.64 -62.64
  TIME         (N_POINTS) datetime64[ns] 2017-07-06T14:49:00 ...

Data variables:
  CONFIG_MISSION_NUMBER (N_POINTS) int64 1 1 1 1 1 ... 3 3 3 3 3 3 3 3
  CYCLE_NUMBER        (N_POINTS) int64 1 1 1 1 1 ... 84 84 84 84 84
  DATA_MODE           (N_POINTS) <U1 'D' 'D' 'D' ... 'D' 'D' 'D' 'D'
  DIRECTION           (N_POINTS) <U1 'D' 'D' 'D' 'D' ... '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
  PRES                (N_POINTS) float64 9.0 14.0 ... 1.488e+03 1.509e+03
  PRES_QC             (N_POINTS) int64 1 1 1 1 1 ... 1 1 1 1 1
  PSAL                (N_POINTS) float64 36.06 36.06 36.06 ... 34.98 34.98
  PSAL_QC             (N_POINTS) int64 1 1 1 1 1 ... 1 1 1 1 1
  TEMP                (N_POINTS) float64 28.04 28.03 28.02 ... 4.281 4.277
  TEMP_QC             (N_POINTS) int64 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

Attributes:
  DATA_ID: ARGO
  DOI: http://doi.org/10.17882/42182
  Fetched_from: https://www.ifremer.fr/erddap
  Fetched_by: docs
  Fetched_date: 2020/11/17
  Fetched_constraints: phy;WMO6902746
  Fetched_uri: ['https://www.ifremer.fr/erddap/tabledap/ArgoFloats...'
  history: Variables filtered according to DATA_MODE; Variable...
```

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

```python
[5]: ds = argo_loader.float([6902746, 6902755]).to_xarray()
print(ds)
```

```
<xarray.Dataset>
Dimensions: (N_POINTS: 18198)
Coordinates:
  * N_POINTS     (N_POINTS) int64 0 1 2 3 ... 18194 18195 18196 18197
  LATITUDE      (N_POINTS) float64 20.08 20.08 20.08 ... 57.96 57.96
```

(continues on next page)
1.5.3 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 to retrieve profiles for, then chain with the `argopy.DataFetcher.to_xarray()` to get the data returned as `xarray.Dataset`.

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

```python
[6]: ds = argo_loader.profile(6902755, 12).to_xarray()
print(ds)
```

```bash
<xarray.Dataset>
Dimensions: (N_POINTS: 107)
Coordinates:
  * N_POINTS (N_POINTS) int64 0 1 2 3 4 5 ... 102 103 104 105 106
  LATITUDE (N_POINTS) float64 63.68 63.68 63.68 ... 63.68 63.68
  LONGITUDE (N_POINTS) float64 -28.81 -28.81 ... -28.81 -28.81
  TIME (N_POINTS) datetime64[ns] 2018-10-19T23:52:00 ... ...
Data variables:
  CONFIG_MISSION_NUMBER (N_POINTS) int64 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 12
  DATA_MODE (N_POINTS) <U1 'R' 'R' 'R' 'R' ... 'R' 'R' 'R' 'R'
  DIRECTION (N_POINTS) <U1 '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
  PRES (N_POINTS) float64 3.0 4.0 ... 1.713e+03 1.732e+03
  PRES_QC (N_POINTS) int64 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
  PSAL_QC (N_POINTS) int64 1 1 1 1 1 1 ... 1 1 1 1 1 1
  TEMP (N_POINTS) float64 7.598 7.599 7.602 ... 3.549 3.536
  TEMP_QC (N_POINTS) int64 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
Attributes:
    DATA_ID: ARGO
    DOI: http://doi.org/10.17882/42182
    Fetched_from: https://www.ifremer.fr/erddap
    Fetched_by: docs
    Fetched_date: 2020/11/17
    Fetched_constraints: phy;WMO6902746;WMO6902755
    Fetched_url: ['https://www.ifremer.fr/erddap/tabledap/ArgoFloats...'
    history: Variables filtered according to DATA_MODE; Variable...
To fetch data for more than one profile, input them in a list:

```python
[7]: ds = argo_loader.profile(6902755, [3, 12]).to_xarray()
print(ds)
```

```python
<xarray.Dataset>
Dimensions: (N_POINTS: 215)
Coordinates:
  * N_POINTS (N_POINTS) int64 0 1 2 3 4 5 ... 210 211 212 213 214
  LATITUDE (N_POINTS) float64 59.72 59.72 59.72 ... 63.68 63.68
  LONGITUDE (N_POINTS) float64 -31.24 -31.24 ... -28.81 -28.81
  TIME (N_POINTS) datetime64[ns] 2018-07-22T00:03:00 ... ...
Data variables:
  CONFIG_MISSION_NUMBER (N_POINTS) int64 2 2 2 2 2 2 ... 2 2 2 2 2 2
  CYCLE_NUMBER (N_POINTS) int64 3 3 3 3 3 3 ... 12 12 12 12 12 12
  DATA_MODE (N_POINTS) <U1 'R' 'R' 'R' 'R' ... 'R' 'R' 'R' 'R'
  DIRECTION (N_POINTS) <U1 '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
  PRES (N_POINTS) float64 3.0 4.0 ... 1.713e+03 1.732e+03
  PRES_QC (N_POINTS) int64 1 1 1 1 1 1 ... 1 1 1 1 1 1
  PSAL (N_POINTS) float64 34.76 34.76 34.76 ... 34.94 34.94
  PSAL_QC (N_POINTS) int64 1 1 1 1 1 1 ... 1 1 1 1 1 1
  TEMP (N_POINTS) float64 8.742 8.743 8.744 ... 3.549 3.536
  TEMP_QC (N_POINTS) int64 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
Attributes:
  DATA_ID: ARGO
  DOI: http://doi.org/10.17882/42182
  Fetched_from: https://www.ifremer.fr/erddap
  Fetched_by: docs
  Fetched_date: 2020/11/17
  Fetched_constraints: phy;WMO6902755_CYC12
  Fetched_uri: ['https://www.ifremer.fr/erddap/tabledap/ArgoFloats...'
  history: Variables filtered according to DATA_MODE; Variable...
```

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

1.6.1 Selecting a source

argopy can get access to Argo data from different sources:

1. the Ifremer erddap server.

   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.

2. your local collection of Argo files, organised as in the GDAC ftp.

   This is how you would use argopy with your data, as long as they are formatted and organised the Argo way. You can select this data source with the keyword localftp and methods described below.

3. the Argovis server.

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

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

- using argopy global options:

  [3]: argopy.set_options(src='erddap')

  [3]: <argopy.options.set_options at 0x7f603c5b24f0>

- in a temporary context:

  [4]: with argopy.set_options(src='erddap'):

      loader = ArgoDataFetcher().profile(6902746, 34)

- with an argument in the data fetcher:

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

1.6.2 Setting a local copy of the GDAC ftp

Data fetching with the localftp data source will require you to specify the path toward your local copy of the GDAC ftp server with the local_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:

If you want to get your own copy of the ftp server content, 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:

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

To synchronize the index:

```bash
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 Comparing data sources

#### Features

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

<table>
<thead>
<tr>
<th>Data source:</th>
<th>erddap</th>
<th>localftp</th>
<th>argovis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Access Points</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>region</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>float</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>profile</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>User mode</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>standard</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>expert</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td><strong>Dataset</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>core (T/S)</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>BGC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference data for DMQC</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Parallel method</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>multi-threading</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>multi-processes</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Dask client</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 source and of your local data.

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()`):

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

# then fetch data:
with argopy.set_options(src='localftp', local_ftp=ftproot):
    ds = ArgoDataFetcher().float(1900857).to_xarray()
print(ds)
```

```
<xarray.Dataset>
Dimensions:  (N_POINTS: 20966)
Coordinates:
    * N_POINTS (N_POINTS) int64 0 1 2 3 ... 20962 20963 20964 20965
    TIME (N_POINTS) datetime64[ns] 2008-02-25T04:03:00 ... ...
    LATITUDE (N_POINTS) float64 -39.93 -39.93 ... -44.16 -44.16
    LONGITUDE (N_POINTS) float64 10.81 10.81 10.81 ... 92.65 92.65
Data variables:
    CONFIG_MISSION_NUMBER (N_POINTS) int64 1 1 1 1 1 1 1 1 ... 2 2 2 2 2 2 2 2
    CYCLE_NUMBER (N_POINTS) int64 0 0 0 0 0 0 0 0 ... 192 192 192 192 192
    DATA_MODE (N_POINTS) <U1 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D' 'D'
    DIRECTION (N_POINTS) <U1 '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
    PRES (N_POINTS) float64 17.0 25.0 ... 1.964e+03 1.987e+03
    PRES_QC (N_POINTS) int64 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1
    PSAL (N_POINTS) float64 34.68 34.68 34.69 ... 34.71 34.72
    PSAL_QC (N_POINTS) int64 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 ... 2.422 2.413
    TEMP_QC (N_POINTS) int64 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
Attributes:
    DATA_ID: ARGO
    DOI: http://doi.org/10.17882/42182
    Fetched_from: /home/docs/.argopy_tutorial_data/ftp
    Fetched_by: docs
    Fetched_date: 2020/11/17
    Fetched_constraints: phy;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:

```python
with argopy.set_options(src='erddap'):
    ds = ArgoDataFetcher().float(1900857).to_xarray()
print(ds)
```

```
<xarray.Dataset>
Dimensions:  (N_POINTS: 20966)
Coordinates:
    * N_POINTS (N_POINTS) int64 0 1 2 3 ... 20962 20963 20964 20965
(continues on next page)
LATITUDE (N_POINTS) float64 -39.93 -39.93 ... -44.16 -44.16
LONGITUDE (N_POINTS) float64 10.81 10.81 10.81 ... 92.65 92.65
TIME (N_POINTS) datetime64[ns] 2008-02-25T04:03:00 ... ...

Data variables:
CONFIG_MISSION_NUMBER (N_POINTS) int64 1 1 1 1 1 1 ... 2 2 2 2 2 2
CYCLE_NUMBER (N_POINTS) int64 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'
DIRECTION (N_POINTS) <U1 'D' 'D' 'D' 'D' ... '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
PRES (N_POINTS) float64 17.0 25.0 ... 1.964e+03 1.987e+03
PRES_QC (N_POINTS) int64 1 1 1 1 1 1 ... 1 1 1 1 1 1
PSAL (N_POINTS) float64 34.68 34.68 34.69 ... 34.71 34.72
PSAL_QC (N_POINTS) int64 1 1 1 1 1 1 ... 1 1 1 1 1 1
TEMP (N_POINTS) float64 16.14 16.14 16.03 ... 2.422 2.413
TIME_QC (N_POINTS) int64 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://www.ifremer.fr/erddap
Fetched_by: docs
Fetched_date: 2020/11/17
Fetched_uri: ['https://www.ifremer.fr/erddap/tabledap/ArgoFloats/...']
import argopy

[8]: with argopy.set_options(src='argovis'):
    ds = ArgoDataFetcher().float(1900857).to_xarray()
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) object '2008-02-28T01:23:00.000Z' ... '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 ... 192 192 192 192 192 192
    DATA_MODE (N_POINTS) <U1 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D'
    DIRECTION (N_POINTS) <U1 'A' 'A' 'A' '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
    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
Attributes:
    DATA_ID: ARGO
    DOI: http://doi.org/10.17882/42182
    Fetched_from: https://argovis.colorado.edu
    Fetched_by: docs
    Fetched_date: 2020/11/17
    Fetched_uri: ['https://argovis.colorado.edu/catalog/platforms/19...']
We can see some minor differences between `localftp/erddap` vs the `argovis` response: this later data source does not include the descending part of the first profile, this explains why `argovis` returns slightly less data.

### 1.6.4 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:

```
src argovis is ok src erddap is ok src localftp is ok
```

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 ressources to the software.

1.7 Manipulating data

Once you fetched data, `argopy` comes with a handy `xarray.Dataset` accessor namespace `argo` to perform specific manipulation of the data.

#### 1.7.1 Transformation

**Points vs profiles**

Fetched data are returned as a 1D array collection of measurements:

```
[3]: argo_loader = ArgoDataFetcher().region([-75,-55,30.,40.,0,100., '2011-01-01', '2011-01-15'])
ds_points = argo_loader.to_xarray()
print(ds_points)
```

```markdown
<xarray.Dataset>
Dimensions:     (N_POINTS: 524)
Coordinates:
    * N_POINTS (N_POINTS) int64 0 1 2 3 4 5 ... 519 520 521 522 523
    LATITUDE   (N_POINTS) float64 37.28 37.28 37.28 ... 33.07 33.07
    LONGITUDE  (N_POINTS) float64 -66.77 -66.77 ... -64.59 -64.59
    TIME       (N_POINTS) datetime64[ns] 2011-01-02T11:14:06 ... ...
Data variables:
    CONFIG_MISSION_NUMBER (N_POINTS) int64 1 1 1 1 1 1 ... 13 13 13 13 13 ...
```

(continues on next page)
If you prefer to work with a 2D array collection of vertical profiles, simply transform the dataset with argopy.

```python
xarray.ArgoAccessor.point2profile()
```

```python
[4]: ds_profiles = ds_points.argo.point2profile()
publish(ds_profiles)
```

<xarray.Dataset>
Dimensions: (N_LEVELS: 50, N_PROF: 18)
Coordinates:
  * N_PROF (N_PROF) int64 7 13 15 6 2 9 ... 12 10 17 3 8 14 16
  * N_LEVELS (N_LEVELS) int64 0 1 2 3 4 5 6 ... 44 45 46 47 48 49
  * LATITUDE (N_PROF) float64 37.28 33.98 32.88 ... 34.39 33.07
  * LONGITUDE (N_PROF) float64 -66.77 -71.17 ... -72.75 -64.59
  * TIME (N_PROF) datetime64[ns] 2011-01-02T11:14:06 ... 20...

Data variables:
  CONFIG_MISSION_NUMBER (N_PROF) int64 1 1 1 1 1 1 0 1 1 1 1 1 1 2 1 1 1 1 13
  CYCLE_NUMBER (N_PROF) int64 150 3 110 180 ... 62 148 151 4 13
  DATA_MODE (N_PROF) <U1 '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 4900803 ... 5903377 5903377
  POSITION_QC (N_PROF) int64 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1
  PRES (N_PROF, N_LEVELS) float64 5.0 10.0 ... 99.97 nan
  PRES_QC (N_PROF) int64 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1
  PSAL (N_PROF, N_LEVELS) float64 36.67 36.67 ... 36.67 nan
  PSAL_QC (N_PROF) int64 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 ... 19.2 nan
  TEMP_QC (N_PROF) int64 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

Attributes:
  DATA_ID: ARGO
  DOI: http://doi.org/10.17882/42182
  Fetched_from: https://www.ifremer.fr/erddap
  Fetched_by: docs
  Fetched_date: 2020/11/17
  Fetched_constraints: [x=-75.00/-55.00; y=30.00/40.00; z=0.0/100.0; t=201...
You can simply reverse this transformation with the `argopyargo.profile2point()`:

```
[5]: ds = ds_profiles.argo.profile2point()
pdprint(ds)
```

```
<xarray.Dataset>
Dimensions: (N_POINTS: 524)
Coordinates:
    LONGITUDE (N_POINTS) float64 -66.77 -66.77 ... -64.59 -64.59
    TIME (N_POINTS) datetime64[ns] 2011-01-02T11:14:06 ... ...
    LATITUDE (N_POINTS) float64 37.28 37.28 37.28 ... 33.07 33.07
    * N_POINTS (N_POINTS) int64 0 1 2 3 4 5 ... 519 520 521 522 523
Data variables:
    CONFIG_MISSION_NUMBER (N_POINTS) int64 1 1 1 1 ... 13 13 13 13 13
    CYCLE_NUMBER (N_POINTS) int64 150 150 150 150 ... 13 13 13 13 13
    DATA_MODE (N_POINTS) <U1 'D' 'D' 'D' ... 'D' 'D' 'D'
    DIRECTION (N_POINTS) <U1 '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
    PRES (N_POINTS) float64 5.0 10.0 15.0 ... 97.97 99.97
    PRES_QC (N_POINTS) int64 1 1 1 1 1 1 ... 1 1 1 1 1
    PSAL (N_POINTS) float64 36.67 36.67 36.67 ... 36.67 36.67
    PSAL_QC (N_POINTS) int64 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
    TEMP_QC (N_POINTS) int64 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
Attributes:
    DATA_ID: ARGO
    DOI: http://doi.org/10.17882/42182
    Fetched_from: https://www.ifremer.fr/erddap
    Fetched_by: docs
    Fetched_date: 2020/11/17
    Fetched_constraints: [x=-75.00/-55.00; y=30.00/40.00; z=0.0/100.0; t=201... hist
```
### Configurations

- **config_mission_number (n_prof)**: float64 1.0 1.0 11.0 1.0 ... 1.0 1.0 13.0
- **cycle_number (n_prof)**: float64 150.0 3.0 11.0 ... 151.0 4.0 13.0
- **data_mode (n_prof)**: object 'D' 'D' 'D' 'D' ... 'D' 'D' 'D' 'D'
- **direction (n_prof)**: object 'A' 'A' 'A' 'A' ... 'A' 'A' 'A' 'A'
- **platform_number (n_prof)**: float64 4.901e+06 4.901e+06 ... 5.903e+06
- **pres (n_prof, pres_interpolated)**: float64 5.0 10.0 ... 50.0
- **psal (n_prof, pres_interpolated)**: float64 36.67 ... 36.68
- **temp (n_prof, pres_interpolated)**: float64 19.46 ... 19.24

### Attributes

- **data_id**: ARGO
- **DOI**: http://doi.org/10.17882/42182
- **fetched_from**: https://www.ifremer.fr/erddap
- **fetched_by**: docs
- **fetched_date**: 2020/11/17
- **fetched_constraints**: [x=-75.00/-55.00; y=30.00/40.00; z=0.0/100.0; t=201...
- **fetched_uri**: ['https://www.ifremer.fr/erddap/tabledap/ArgoFloats...

### 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.

### Filters

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

[To be added]

### 1.7.2 Complementary data

#### TEOS-10 variables

You can compute additional ocean variables from TEOS-10. The default list (and available right now) of variables is: ‘SA’, ‘CT’, ‘SIG0’, ‘N2’, ‘PV’, ‘PTEMP’. Simply raise an issue to add a new one.

This can be done using the `argopy.xarray.ArgoAccessor.teos10()` method and indicating the list of variables you want to compute:

```
[7]: ds = ArgoDataFetcher().float(2901623).to_xarray()
    ds.arro.teos10(['SA', 'CT', 'PV'])
    print(ds)
```
<xarray.Dataset>
Dimensions: (N_POINTS: 8339)
Coordinates:
  * N_POINTS (N_POINTS) int64 0 1 2 3 4 ... 8335 8336 8337 8338
  LATITUDE (N_POINTS) float64 0.012 0.012 0.012 ... 3.388 3.388
  LONGITUDE (N_POINTS) float64 92.28 92.28 92.28 ... 94.77 94.77
  TIME (N_POINTS) datetime64[ns] 2010-05-14T03:35:00 ... ...
Data variables:
  CONFIG_MISSION_NUMBER (N_POINTS) int64 1 1 1 1 1 1 ... 1 1 1 1 1 1
  CYCLE_NUMBER (N_POINTS) int64 0 0 0 0 0 0 ... 96 96 96 96 96
  DATA_MODE (N_POINTS) <U1 'R' 'R' 'R' 'R' ... 'R' 'R' 'R' 'R'
  DIRECTION (N_POINTS) <U1 'D' 'D' 'D' 'D' ... '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
  PRES (N_POINTS) float64 17.0 25.0 ... 1.112e+03 1.137e+03
  PRES_QC (N_POINTS) int64 1 1 1 1 1 1 ... 1 1 1 1 1 1
  PSAL (N_POINTS) float64 34.27 34.28 34.28 ... 34.92 34.91
  PSAL_QC (N_POINTS) float64 1 1 1 1 1 1 ... 1 1 1 1 1 1
  TEMP (N_POINTS) float64 30.16 30.17 30.17 ... 6.189 6.071
  TEMP_QC (N_POINTS) int64 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
  SA (N_POINTS) float64 34.44 34.44 34.44 ... 35.09 35.08
  CT (N_POINTS) float64 30.21 30.21 30.22 ... 6.269 6.152
  PV (N_POINTS) float64 -6.868e-15 ... 1.435e-12 nan
Attributes:
  DATA_ID: ARGO
  DOI: http://doi.org/10.17882/42182
  Fetched_from: https://www.ifremer.fr/erddap
  Fetched_by: docs
  Fetched_date: 2020/11/17
  Fetched_constraints: phy;WMO2901623
  Fetched_uri: ['https://www.ifremer.fr/erddap/tabledap/ArgoFloats...]
  history: Variables filtered according to DATA_MODE; Variable...

[8]: print(ds['SA'])

<xarray.DataArray 'SA' (N_POINTS: 8339)>
array([34.43589931, 34.43691224, 34.43692096, ..., 35.0921157 ,
       35.09227648, 35.08238554])
Coordinates:
  * N_POINTS (N_POINTS) int64 0 1 2 3 4 5 6 ... 8333 8334 8335 8336 8337 8338
  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 ... ...
Attributes:
  standard_name: Absolute Salinity
  unit: g/kg
1.7.3 Data models

By default argopy works with xarray.DataSet and comes with the xarray.Dataset accessor namespace argo. For your own analysis, you may prefer to work with a Pandas dataframe.

```python
df = ArgoDataFetcher().profile(6902746, 34).to_dataframe()
df
```

<table>
<thead>
<tr>
<th>CONFIG_MISSION_NUMBER</th>
<th>CYCLE_NUMBER</th>
<th>DATA_MODE</th>
<th>DIRECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>34</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>34</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>34</td>
<td>D</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>34</td>
<td>D</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>34</td>
<td>A</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>104</td>
<td>2</td>
<td>34</td>
<td>D</td>
</tr>
<tr>
<td>105</td>
<td>2</td>
<td>34</td>
<td>D</td>
</tr>
<tr>
<td>106</td>
<td>2</td>
<td>34</td>
<td>D</td>
</tr>
<tr>
<td>107</td>
<td>2</td>
<td>34</td>
<td>D</td>
</tr>
<tr>
<td>108</td>
<td>2</td>
<td>34</td>
<td>D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PLATFORM_NUMBER</th>
<th>POSITION_QC</th>
<th>PRES</th>
<th>PRES_QC</th>
<th>PSAL</th>
<th>PSAL_QC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6902746</td>
<td>1</td>
<td>3.0</td>
<td>1</td>
<td>36.262001</td>
</tr>
<tr>
<td>1</td>
<td>6902746</td>
<td>1</td>
<td>4.0</td>
<td>1</td>
<td>36.262001</td>
</tr>
<tr>
<td>2</td>
<td>6902746</td>
<td>1</td>
<td>5.0</td>
<td>1</td>
<td>36.263000</td>
</tr>
<tr>
<td>3</td>
<td>6902746</td>
<td>1</td>
<td>6.0</td>
<td>1</td>
<td>36.262001</td>
</tr>
<tr>
<td>4</td>
<td>6902746</td>
<td>1</td>
<td>7.0</td>
<td>1</td>
<td>36.262001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>104</td>
<td>6902746</td>
<td>1</td>
<td>1913.0</td>
<td>1</td>
<td>34.976002</td>
</tr>
<tr>
<td>105</td>
<td>6902746</td>
<td>1</td>
<td>1938.0</td>
<td>1</td>
<td>34.980999</td>
</tr>
<tr>
<td>106</td>
<td>6902746</td>
<td>1</td>
<td>1964.0</td>
<td>1</td>
<td>34.984001</td>
</tr>
<tr>
<td>107</td>
<td>6902746</td>
<td>1</td>
<td>1988.0</td>
<td>1</td>
<td>34.983002</td>
</tr>
<tr>
<td>108</td>
<td>6902746</td>
<td>1</td>
<td>2007.0</td>
<td>1</td>
<td>34.983002</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TEMP</th>
<th>TEMP_QC</th>
<th>TIME_QC</th>
<th>LATITUDE</th>
<th>LONGITUDE</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>27.212000</td>
<td>1</td>
<td>1</td>
<td>18.983</td>
<td>-58.119 2017-12-20 06:58:00</td>
</tr>
<tr>
<td>1</td>
<td>27.212000</td>
<td>1</td>
<td>1</td>
<td>18.983</td>
<td>-58.119 2017-12-20 06:58:00</td>
</tr>
<tr>
<td>2</td>
<td>27.212999</td>
<td>1</td>
<td>1</td>
<td>18.983</td>
<td>-58.119 2017-12-20 06:58:00</td>
</tr>
<tr>
<td>3</td>
<td>27.212000</td>
<td>1</td>
<td>1</td>
<td>18.983</td>
<td>-58.119 2017-12-20 06:58:00</td>
</tr>
<tr>
<td>4</td>
<td>27.214001</td>
<td>1</td>
<td>1</td>
<td>18.983</td>
<td>-58.119 2017-12-20 06:58:00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>104</td>
<td>3.710000</td>
<td>1</td>
<td>1</td>
<td>18.983</td>
<td>-58.119 2017-12-20 06:58:00</td>
</tr>
<tr>
<td>105</td>
<td>3.718000</td>
<td>1</td>
<td>1</td>
<td>18.983</td>
<td>-58.119 2017-12-20 06:58:00</td>
</tr>
<tr>
<td>106</td>
<td>3.698000</td>
<td>1</td>
<td>1</td>
<td>18.983</td>
<td>-58.119 2017-12-20 06:58:00</td>
</tr>
<tr>
<td>107</td>
<td>3.668000</td>
<td>1</td>
<td>1</td>
<td>18.983</td>
<td>-58.119 2017-12-20 06:58:00</td>
</tr>
<tr>
<td>108</td>
<td>3.644000</td>
<td>1</td>
<td>1</td>
<td>18.983</td>
<td>-58.119 2017-12-20 06:58:00</td>
</tr>
</tbody>
</table>

[109 rows x 16 columns]

but keep in mind that this is merely a short cut for the `xarray.Dataset.to_dataframe()` method.
1.8 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().

```python
import argopy
from argopy import DataFetcher as ArgoDataFetcher
```

1.9 User mode: standard vs expert

Problem

For beginners or 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 standard users, we implemented in argopy different levels of verbosity and data processing to hide or simply remove variables only meaningful to experts.

1.9.1 What type of user are you ?

If you don’t know in which user 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 answered to no more than 1 question, you probably would feel more comfortable with the standard user mode. Otherwise, you can give a try to the expert mode.

In standard mode, fetched data are automatically filtered to account for their quality (only good are retained) and level of processing by the data centers (whether they looked at the data briefly or not).
1.9.2 Setting the user mode

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

If you want to change the user mode, or simply makes it explicit, you can use:

- argopy global options:

```python
[3]: argopy.set_options(mode='standard')
[3]: <argopy.options.set_options at 0x7f39dd0fda60>
```

- a temporary context:

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

- option when instantiating the data fetcher:

```python
[5]: ArgoDataFetcher(mode='standard').profile(6902746, 34)
[5]: <datafetcher.erddap>
Name: Ifremer erddap Argo data fetcher for floats
API: https://www.ifremer.fr/erddap
Domain: phy;WMO6902746_CYC34
Backend: erddap
User mode: standard
```

1.9.3 Differences in user modes

To highlight that, let’s compare data fetched for one profile with each modes.

You will note that the standard mode has fewer variables to let you focus on your analysis. For expert, all Argo variables for you to work with are here.

The difference is the most visible when fetching Argo data from a local copy of the GDAC ftp, so let’s use a sample of this provided by argopy tutorial datasets:

```python
[6]: ftproot, flist = argopy.tutorial.open_dataset('localftp')
argopy.set_options(local_ftp=ftproot)
[6]: <argopy.options.set_options at 0x7f39b5f69fd0>
```

In standard mode:

```python
[7]: with argopy.set_options(mode='standard'):
    ds = ArgoDataFetcher(src='localftp').profile(6901929, 2).to_xarray()
    print(ds.data_vars)
```

Data variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONFIG_MISSION_NUMBER</td>
<td>(N_POINTS) int64</td>
</tr>
<tr>
<td>CYCLE_NUMBER</td>
<td>(N_POINTS) int64</td>
</tr>
<tr>
<td>DATA_MODE</td>
<td>(N_POINTS) &lt;U1</td>
</tr>
<tr>
<td>DIRECTION</td>
<td>(N_POINTS) &lt;U1</td>
</tr>
<tr>
<td>PLATFORM_NUMBER</td>
<td>(N_POINTS) int64</td>
</tr>
<tr>
<td>POSITION_QC</td>
<td>(N_POINTS) int64</td>
</tr>
<tr>
<td>PRES</td>
<td>(N_POINTS) float64</td>
</tr>
<tr>
<td>PRES_QC</td>
<td>(N_POINTS) int64</td>
</tr>
<tr>
<td>PSAL</td>
<td>(N_POINTS) float64</td>
</tr>
</tbody>
</table>

(continues on next page)
In expert mode:

```python
[8]: with argopy.set_options(mode='expert'):
  ds = ArgoDataFetcher(src='localftp').profile(6901929, 2).to_xarray()
  print(ds.data_vars)
```

<table>
<thead>
<tr>
<th>Data variables:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONFIGMISSIONNUMBER (N_POINTS) int64 2 2</td>
</tr>
<tr>
<td>CYCLENUMBER (N_POINTS) int64 2 2</td>
</tr>
<tr>
<td>DATACENTRE (N_POINTS) &lt;U2 'BO' 'BO'</td>
</tr>
<tr>
<td>DATAMODE (N_POINTS) &lt;U1 'R' 'R'</td>
</tr>
<tr>
<td>DATASTATEINDICATOR (N_POINTS) &lt;U2 '2B ' '2B '</td>
</tr>
<tr>
<td>DCREFERENCE (N_POINTS) &lt;U32 ' '</td>
</tr>
<tr>
<td>DIRECTION (N_POINTS) &lt;U1 'A' 'A'</td>
</tr>
<tr>
<td>FIRMWAREVERSION (N_POINTS) &lt;U32 '5900A04'</td>
</tr>
<tr>
<td>FLOATSERIALNO (N_POINTS) &lt;U32 'AI2600-17EU01'</td>
</tr>
<tr>
<td>PI_NAME (N_POINTS) &lt;U64 'Diarmuid O’Conchubhair'</td>
</tr>
<tr>
<td>PLATFORMNUMBER (N_POINTS) int64 6901929 6901929</td>
</tr>
<tr>
<td>PLATFORMTYPE (N_POINTS) &lt;U32 'ARVOR '</td>
</tr>
<tr>
<td>POSITIONINGSYSTEM (N_POINTS) &lt;U8 'GPS ' 'GPS '</td>
</tr>
<tr>
<td>POSITIONQC (N_POINTS) &lt;U1 'A' 'A'</td>
</tr>
<tr>
<td>PRES (N_POINTS) float32 0.4 0.5</td>
</tr>
<tr>
<td>PRESADJUSTED (N_POINTS) float32 nan nan</td>
</tr>
<tr>
<td>PRESADJUSTEDERROR (N_POINTS) float32 nan nan</td>
</tr>
<tr>
<td>PRES_ADJUSTED_QC (N_POINTS) int64 0 0</td>
</tr>
<tr>
<td>PRES_QC (N_POINTS) int64 1 1</td>
</tr>
<tr>
<td>PSAL (N_POINTS) float32 35.342 35.343</td>
</tr>
<tr>
<td>PSALADJUSTED (N_POINTS) float32 nan nan</td>
</tr>
<tr>
<td>PSALADJUSTEDERROR (N_POINTS) float32 nan nan</td>
</tr>
<tr>
<td>PSALPRODUCT (N_POINTS) int64 0 0</td>
</tr>
<tr>
<td>PSALQC (N_POINTS) int64 1 1</td>
</tr>
<tr>
<td>TEMP (N_POINTS) float32 10.028 10.028</td>
</tr>
<tr>
<td>TEMPADJUSTED (N_POINTS) float32 nan nan</td>
</tr>
<tr>
<td>TEMPADJUSTEDERROR (N_POINTS) float32 nan nan</td>
</tr>
<tr>
<td>TEMP_ADJUSTED_QC (N_POINTS) int64 0 0</td>
</tr>
<tr>
<td>TEMP_QC (N_POINTS) int64 4 4</td>
</tr>
<tr>
<td>TIMELOCATION (N_POINTS) datetime64[ns] 2018-02-24T09:16:24.9...</td>
</tr>
<tr>
<td>TIMEQC (N_POINTS) int64 1 1</td>
</tr>
<tr>
<td>VERTICAL_SAMPLING_SCHEME (N_POINTS) &lt;U256 'Primary sampling: averaged [1...</td>
</tr>
<tr>
<td>WMOINSTTYPE (N_POINTS) int64 844 844</td>
</tr>
</tbody>
</table>
## 1.10 Fetching Argo meta-data

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` has a specific fetcher for index:

```python
from argopy import IndexFetcher as ArgoIndexFetcher
index_loader = ArgoIndexFetcher()
```

```python
idx = index_loader.region([[-75, -45, 20, 30], '2011-01-01', '2011-05'])
idx
```

```
<indexfetcher.erddap>
Name: Ifremer erddap Argo Index fetcher for a space/time region
API: https://www.ifremer.fr/erddap
Domain: [x=-75.00/-45.00; y=20.00/30.00; t=2011-01-01/2011-05-01]
Backend: erddap
User mode: standard
```

```python
idx.to_dataframe()
```

```
<table>
<thead>
<tr>
<th>file</th>
<th>date</th>
<th>longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>aoml/4900753/profiles/D4900753_176.nc</td>
<td>2011-01-01 16:32:09</td>
<td>-68.188</td>
</tr>
<tr>
<td>aoml/4901211/profiles/D4901211_010.nc</td>
<td>2011-01-02 10:10:50</td>
<td>-51.584</td>
</tr>
<tr>
<td>aoml/4900818/profiles/D4900818_135.nc</td>
<td>2011-01-02 10:23:10</td>
<td>-60.819</td>
</tr>
<tr>
<td>aoml/1901461/profiles/D1901461_023.nc</td>
<td>2011-01-02 11:26:40</td>
<td>-56.303</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>aoml/4900780/profiles/D4900780_172.nc</td>
<td>2011-04-30 12:02:21</td>
<td>-60.472</td>
</tr>
<tr>
<td>coriolis/6900778/profiles/D6900778_007.nc</td>
<td>2011-04-30 15:51:14</td>
<td>-64.979</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>file</th>
<th>date</th>
<th>longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

```python
idx.to_dataframe()
```

```
<table>
<thead>
<tr>
<th>file</th>
<th>date</th>
<th>longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>aoml/4900753/profiles/D4900753_176.nc</td>
<td>2011-01-01 16:32:09</td>
<td>-68.188</td>
</tr>
<tr>
<td>aoml/4901211/profiles/D4901211_010.nc</td>
<td>2011-01-02 10:10:50</td>
<td>-51.584</td>
</tr>
<tr>
<td>aoml/4900818/profiles/D4900818_135.nc</td>
<td>2011-01-02 10:23:10</td>
<td>-60.819</td>
</tr>
<tr>
<td>aoml/1901461/profiles/D1901461_023.nc</td>
<td>2011-01-02 11:26:40</td>
<td>-56.303</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>aoml/4900780/profiles/D4900780_172.nc</td>
<td>2011-04-30 12:02:21</td>
<td>-60.472</td>
</tr>
<tr>
<td>coriolis/6900778/profiles/D6900778_007.nc</td>
<td>2011-04-30 15:51:14</td>
<td>-64.979</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>file</th>
<th>date</th>
<th>longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

(continues on next page)
1.11 Data visualisation

Although argopy focus is not on visualisation, it provides a few functions to get you started. A gallery of examples is in preparation for a future release.

1.11.1 Trajectories from an index

```python
from argopy import IndexFetcher as ArgoIndexFetcher
idx = ArgoIndexFetcher().float([6902745, 6902746])
idx.to_dataframe()
idx.plot('trajectory')
```

1.11.2 Float dashboard

When working in Jupyter notebook, you can insert the EuroArgo dashboard in a cell with:

```python
import argopy
argopy.dashboard()
```
and for a specific float, just provide its WMO:

```python
import argopy
argopy.dashboard(wmo=6902746)
```

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

### 1.11. Data visualisation
1.12 Performance

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,
- Fetch data by chunks in parallel, i.e. fetch peace of independant 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).

1.12.1 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'):
  
  ds = ArgoDataFetcher(cache=True).profile(6902746, 34).to_xarray()

- when instantiating the data fetcher:

  ds = ArgoDataFetcher(cache=True, cachedir='mycache_folder').profile(6902746, 34).to_xarray()


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:

```python
cache = ArgoDataFetcher(cache=True, cachedir='mycache_folder').profile(6902746, 34)
cache.to_xarray();
```

Fetched data are in the local cache folder:

```python
os.listdir('mycache_folder')
```

```
['cache',
 'c5c820b6aff7b2ef86ef00626782587a95d37edc54120a63ee4699be2b0c6b7c']
```

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

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

```python
fetcher2 = ArgoDataFetcher(cache=True, cachedir='mycache_folder').profile(1901393, 1)
fetcher2.to_xarray();
```

All fetched data are cached:

```python
os.listdir('mycache_folder')
```

```
['cache',
 'c5c820b6aff7b2ef86ef00626782587a95d37edc54120a63ee4699be2b0c6b7c',
 '58072df8477157c194449a2e6dff8d69ca3c8f6ed01eebddd8a5fc446f2f7f9a7']
```

Note the new hash file with the fetcher2 data.

It is important to note that we can safely clear the cache from the first fetcher data, it won’t remove the fetcher2 data:

```python
fetcher.clear_cache()
```

```python
os.listdir('mycache_folder')
```

```
['cache',
 '58072df8477157c194449a2e6dff8d69ca3c8f6ed01eebddd8a5fc446f2f7f9a7']
```

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:

```python
argopy.clear_cache()
```

So, if we now check the cache folder, it’s been deleted:

```python
os.listdir('mycache_folder')
```

```
---------------------------------------------------------------------------
FileNotFoundError Traceback (most recent call last)
<ipython-input-13-6726e674f21f> in <module>
     462
     463
<ipython-input-13-6726e674f21f> in <module>
     462
     463
```

(continues on next page)
1.12.2 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:

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

# Instantiate a parallel fetcher:
loader_par = ArgoDataFetcher(src='erddap', parallel=True).region(box)
```

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

```python
loader_par = ArgoDataFetcher(src='erddap', parallel=True, progress=True).region(box)
```

Then, you can fetch data as usual:

```python
%%time
ds = loader_par.to_xarray()
```

**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:

```python
for uri in loader_par.uri:
    print("http: ... ", "&".join(uri.split("&")[1:-2]))  # Display only the relevant part of each URLs of URI:
```

```text
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.

```python
[7]: Create a large box:
box = [-60, 0, 0.0, 60.0, 0.0, 500.0, "2007", "2010"]

# Init a parallel fetcher:
loader_par = ArgoDataFetcher(src='erddap',
                             parallel=True,
                             chunks={'lon': 5}).region(box)

# Check number of chunks:
len(loader_par.uri)

[7]: 195
```

This creates 195 chunks, and 5 along the longitudinal 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:

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

# Check number of chunks:
len(loader_par.uri)

[8]: 5
```

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

```python
[9]: for uri in loader_par.uri:
    print("&".join(uri.split("&")[:2])) # Display only the relevant URL part
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 `localftp` 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:

<table>
<thead>
<tr>
<th>Access point dimension</th>
<th>Maximum chunk size</th>
</tr>
</thead>
<tbody>
<tr>
<td>region / lon</td>
<td>20 deg</td>
</tr>
<tr>
<td>region / lat</td>
<td>20 deg</td>
</tr>
<tr>
<td>region / dpt</td>
<td>500 m or db</td>
</tr>
<tr>
<td>region / time</td>
<td>90 days</td>
</tr>
<tr>
<td>float / wmo</td>
<td>5</td>
</tr>
<tr>
<td>profile / wmo</td>
<td>5</td>
</tr>
</tbody>
</table>

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:

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

# Init a parallel fetcher:
loader_par = ArgoDataFetcher(src='erddap',
                              parallel=True,
                              chunks_maxsize={'dpt': 100}).region(box)

# Check number of chunks:
len(loader_par.uri)
```

[10]: 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:

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

for uri in loader_par.uri:
    print("http: ... ", ", ".join(uri.split("&")[:-2]))
```

(continues on next page)
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.

```python
[12]:
WMO_list = [6902766, 6902772, 6902914, 6902746, 6902916, 6902915, 6902757, 6902771]

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

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.

**Note:** 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 `localftp`.

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:

<table>
<thead>
<tr>
<th>Parallel method</th>
<th>erddap</th>
<th>localftp</th>
<th>argovis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-threading</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Multi-processes</td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

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

ArgoDataFetcher(parallel=True, parallel_method='thread')
ArgoDataFetcher(parallel='thread')
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 of data will be requested and not cached by server.

```
[13]: def this_box():
    return [-60, 0,
            20.0, 60.0 + np.random.randint(0,100,1)[0]/1000,
            0.0, 500.0,
            "2007", "2009"]

[14]: %time
    b1 = this_box()
    f1 = ArgoDataFetcher(src='argovis', parallel=False).region(b1)
    ds1 = f1.to_xarray()
CPU times: user 16.7 s, sys: 312 ms, total: 17 s
Wall time: 29 s

[15]: %time
    b2 = this_box()
    f2 = ArgoDataFetcher(src='argovis', parallel=True).region(b2)
    ds2 = f2.to_xarray()
CPU times: user 20.9 s, sys: 624 ms, total: 21.5 s
Wall time: 23.2 s
```

This simple comparison 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:

  ```python
  ArgoDataFetcher(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
1.13 What's New

1.13.1 v0.1.7 (XX Nov. 2020)

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.

```python
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.

```python
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

- 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, preprocess 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.

1.13.2 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.


1.13.3 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.

```python
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.

```python
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 point 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).

### 1.13.4 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).
### 1.13.5 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.

```python
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)

### 1.13.6 v0.1.2 (15 May 2020)

We didn’t like this one this morning, so we move one to the next one!
1.13.7 v0.1.1 (3 Apr. 2020)

Features and front-end API

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

1.13.8 v0.1.0 (17 Mar. 2020)

• Initial release.
• Erddap data fetcher

1.14 Contributing to argopy

Table of contents:

• Where to start?
• Bug reports and enhancement requests
• Contributing to the documentation
  – About the argopy documentation
  – How to build the argopy documentation
    * Requirements
    * Building the documentation
• Working with the code
  – Development workflow
  – Virtual environment
  – Code standards
  – Code Formatting
• Contributing to the code base
  – Data fetchers
    * Introduction
    * Detailed guideline
      • Inheritance
      • Auto-discovery of fetcher properties
      • Auto-discovery of fetcher access points
      • File systems
      • Output data format

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 the chat room at gitter.
1.14.1 Where to start?

All contributions, bug reports, bug fixes, documentation improvements, enhancements, and ideas are welcome. We will complete this document for guidelines with regard to each of these contributions over time.

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 issue 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 the chat room at gitter.

1.14.2 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:

```python
>>> 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

1.14.3 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
- How to build the argopy 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](https://sphinx-doc.org). 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](https://numpy.org/doc/stable/user/niumpy/niumpy.html) for a detailed explanation, or look at some of the existing functions to extend it in a similar manner.

- The tutorials make heavy 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](#), but to build the docs you need to use the specific file `docs/requirements.txt`:

```
$ conda create --yes -n argopy-docs python=3.6 xarray dask numpy pytest future gsw
   --sphinx sphinx_rtd_theme
$ conda activate argopy-docs
```

(continues on next page)
$ pip install argopy
$ 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

1.14.4 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.
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

This is how to create a virtual environment into which to test-install argopy, install it, check the version, and tear down the virtual environment.

```
$ conda create --yes -n argopy-tests python=3.6 xarray dask numpy pytest future gsw
$ conda activate argopy-tests
$ pip install argopy
$ python -c 'import argopy; print(argopy.__version__);'
$ conda deactivate
$ conda env remove --yes -n argopy-tests
```

### 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.
1.14.5 Contributing to the code base

Code Base:

- **Data fetchers**
  - **Introduction**
  - **Detailed guideline**
    - *Inheritance*
    - *Auto-discovery of fetcher properties*
    - *Auto-discovery of fetcher access points*
    - *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’, ‘bfc’]`
  - 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.:

```python
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:

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

It could also be like:

```python
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).
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:

```python
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.

1.15 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.

1.15.1 Top-levels functions

<table>
<thead>
<tr>
<th><code>argopy.DataFetcher</code></th>
<th>alias of <code>argopy.fetchers.ArgoDataFetcher</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>argopy.IndexFetcher</code></td>
<td>alias of <code>argopy.fetchers.ArgoIndexFetcher</code></td>
</tr>
</tbody>
</table>

`argopy.DataFetcher`

`DataFetcher`

alias of `argopy.fetchers.ArgoDataFetcher`
argopy.IndexFetcher

IndexFetcher
   alias of argopy.fetchers.ArgoIndexFetcher

Fetching entries

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>argopy.DataFetcher.region(box)</td>
<td>Space/time domain data fetcher</td>
</tr>
<tr>
<td>argopy.DataFetcher.float(wmo, **kw)</td>
<td>Float data fetcher</td>
</tr>
<tr>
<td>argopy.DataFetcher.profile(wmo, cyc)</td>
<td>Specific profile data fetcher</td>
</tr>
</tbody>
</table>

argopy.DataFetcher.region

DataFetcher.region(box: list)
   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

argopy.fetchers.ArgoDataFetcher A data source fetcher for a space/time domain

argopy.DataFetcher.float

DataFetcher.float(wmo, **kw)
   Float data 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

argopy.fetchers.ArgoDataFetcher.float A data source fetcher for all float profiles
**argopy.DataFetcher.profile**

`DataFetcher.profile(wmo, cyc)`

Specific profile data fetcher

**Parameters**

- **wmo**: list(int)  Define the list of Argo floats to load data for. This is a list of integers with WMO numbers.
- **cyc**: list(int)  Define the list of cycle numbers to load for each Argo floats listed in `wmo`.

**Returns**

`argopy.fetchers.ArgoDataFetcher.profile` A data source fetcher for specific float profiles

---

**argopy.IndexFetcher.region**

`IndexFetcher.region(box)`

Load index for a rectangular space/time domain region

**argopy.IndexFetcher.float**

`IndexFetcher.float(wmo)`

Load index for one or more WMOs

---

**I/O and Data formats**

**argopy.DataFetcher.to_xarray(**kwargs)**  Fetch and return data as xarray.DataSet

**argopy.DataFetcher.to_dataframe(**kwargs)**  Fetch and return data as pandas.DataFrame

---

**argopy.DataFetcher.to_xarray**

`DataFetcher.to_xarray(**kwargs)`

Fetch and return data as xarray.DataSet

**Returns**

`xarray.DataArray`
argopy.DataFetcher.to_dataframe

DataFetcher.to_dataframe(**kwargs)
Fetch and return data as pandas.DataFrame

argopy.IndexFetcher.to_xarray(**kwargs)
Fetch index and return xr.dataset

argopy.IndexFetcher.to_dataframe(**kwargs)
Fetch index and return pandas.DataFrame

argopy.IndexFetcher.to_csv(file)
Fetch index and return csv

Visualisation

argopy.IndexFetcher.plot(ptype)
Create custom plots from index

argopy.dashboard(wmo, cyc, width, height, . . .)
Insert in a notebook the Euro-Argo dashboard page

argopy.IndexFetcher.plot

IndexFetcher.plot(ptype='trajectory')
Create custom plots from index

Parameters

ptype: str Type of plot to generate. This can be: ‘trajectory’, ‘profiler’, ‘dac’.

Returns

fig [matplotlib.pyplot.figure.Figure] Figure instance
argopy Documentation, Release 999

argopy.dashboard

```
dashboard(wmo=None, cyc=None, width='100%', height=1000, url=None, type='ea')
```

Insert in a notebook the Euro-Argo dashboard page

**Parameters**

- **wmo**: int The float WMO to display. By default, this is set to None and will insert the general dashboard.

**Returns**

- IFrame: IPython.lib.display.IFrame

**Fetcher properties**

```
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(str)

**Helpers**

```
argopy.set_options(**kwargs)
argopy.clear_cache()
argopy.tutorial.open_dataset(name)
```

**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, localftp, argovis
- **local_ftp**: Absolute path to a local GDAC ftp copy. Default: 
- **cachedir**: Absolute path to a local cache directory. Default: ~/.cache/argopy
- **mode**: User mode. Default: standard. Possible values: standard or expert.
- **api_timeout**: Define the time out of internet requests to web API, in seconds. Default: 120
You can use `set_options` either as a context manager: >>> import argopy >>> with argopy.set_options(src='localftp'): >>> ds = argopy.DataFetcher().float(3901530).to_xarray()

Or to set global options: >>> argopy.set_options(src='localftp')

```python
__init__(**kwargs)
    Initialize self. See help(type(self)) for accurate signature.
```

**Methods**

```python
__init__(**kwargs)
    Initialize self.
```

argopy.clear_cache

clear_cache()
    Delete argopy cache folder content

argopy.tutorial.open_dataset

tutorial.open_dataset(name)
    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: localftp, weekly_index_prof or global_index_prof.
    • localftp, return the absolute path and list of files in the sample local ftp files.
    • weekly_index_prof, return path and to weekly profile index file
    • global_index_prof, return path and to global profile index file

Returns
    path: str Root path to files
    files: list(str) or str List of files with the requested dataset

1.15.2 Low-level functions

```python
argopy.show_versions([file])
    Print the versions of argopy and its dependencies

argopy.utilities.
list_available_data_src()
    List all available data sources

argopy.utilities.
list_available_index_src()
    List all available index sources
```
argopy.show_versions

`show_versions(file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>)`

Print the versions of argopy and its dependencies

**Parameters**

- **file** [file-like, optional] print to the given file-like object. Defaults to sys.stdout.

argopy.utilities.list_available_data_src

`list_available_data_src()`

List all available data sources

argopy.utilities.list_available_index_src

`list_available_index_src()`

List all available index sources

1.15.3 Internals

File systems

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>argopy.stores.filestore(cache, cachedir, ...)</code></td>
<td>Argo local file system</td>
</tr>
<tr>
<td><code>argopy.stores.httpstore(cache, cachedir, ...)</code></td>
<td>Argo http file system</td>
</tr>
<tr>
<td><code>argopy.stores.memorystore(cache, cachedir, ...)</code></td>
<td>Argo in-memory file system</td>
</tr>
</tbody>
</table>

argopy.stores.filestore

```
class filestore(cache: bool = False, cachedir: str = '', **kwargs)
```

Argo local file system

Relies on: https://filesystem-spec.readthedocs.io/en/latest/api.html#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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>([cache, cachedir])</td>
<td>Create a file storage system for Argo data</td>
</tr>
<tr>
<td>cachepath(uri[, errors])</td>
<td>Return path to cached file for a given URI</td>
</tr>
<tr>
<td>clear_cache()</td>
<td>Remove cache files and entry from uri open with this store instance</td>
</tr>
<tr>
<td>exists(path, *args)</td>
<td></td>
</tr>
<tr>
<td>glob(path, **kwargs)</td>
<td></td>
</tr>
<tr>
<td>open(path, *args, **kwargs)</td>
<td></td>
</tr>
<tr>
<td>open_dataset(url, *args, **kwargs)</td>
<td>Return a xarray.dataset from an url</td>
</tr>
<tr>
<td>open_mfdataset(urls[, concat_dim, ...])</td>
<td>Open multiple urls as a single xarray dataset.</td>
</tr>
<tr>
<td>read_csv(url, **kwargs)</td>
<td>Return a pandas.dataframe from an url that is a csv resource</td>
</tr>
<tr>
<td>register(uri)</td>
<td>Keep track of files open with this instance</td>
</tr>
<tr>
<td>store_path(uri)</td>
<td></td>
</tr>
</tbody>
</table>

### Attributes

- **protocol**: argopy.stores.httpstore

#### argopy.stores.httpstore

**class httpstore**(cache: bool = False, cachedir: str = '', **kwargs)

Argo http file system


This store intends to make argopy: safer to failures from http requests, provide more verbose message to users if we can identify specific errors in http responses.

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
argopy Documentation, Release 999

## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>Create a file storage system for Argo data</td>
</tr>
<tr>
<td>cachepath</td>
<td>Return path to cached file for a given URI</td>
</tr>
<tr>
<td>clear_cache</td>
<td>Remove cache files and entry from uri open with this store instance</td>
</tr>
<tr>
<td>exists</td>
<td>Check if file exists at the given path with additional arguments/kwargs</td>
</tr>
<tr>
<td>glob</td>
<td>Return files matching the given path with additional arguments/kwargs</td>
</tr>
<tr>
<td>open</td>
<td>Open and decode an xarray dataset from an url with additional arguments/kwargs</td>
</tr>
<tr>
<td>open_json</td>
<td>Return a json from an url with additional arguments/kwargs or raise errors</td>
</tr>
<tr>
<td>open_mfdataset</td>
<td>Open multiple urls as a single xarray dataset with additional arguments/kwargs</td>
</tr>
<tr>
<td>open_mfjson</td>
<td>Open multiple json urls with additional arguments/kwargs</td>
</tr>
<tr>
<td>read_csv</td>
<td>Read a comma-separated values (csv) url into Pandas DataFrame</td>
</tr>
<tr>
<td>register</td>
<td>Keep track of files open with this instance</td>
</tr>
<tr>
<td>store_path</td>
<td>Return path to file with additional arguments/kwargs</td>
</tr>
</tbody>
</table>

## Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protocol</td>
<td>argopy.stores.memorystore</td>
</tr>
</tbody>
</table>

### argopy.stores.memorystore

**Class** memorystore

```python
class memorystore:
    def __init__(cache: bool = False, cachedir: str = '', **kwargs):
        Argo in-memory file system
        Note that this inherits from filestore, not argo_store_proto
        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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>([cache, cachedir])</td>
<td>Create a file storage system for Argo data</td>
</tr>
<tr>
<td>cachepath(uri[, errors])</td>
<td>Return path to cached file for a given URI</td>
</tr>
<tr>
<td>clear_cache()</td>
<td>Remove cache files and entry from uri open with this store instance</td>
</tr>
<tr>
<td>exists(path, *args)</td>
<td></td>
</tr>
<tr>
<td>glob(path, **kwargs)</td>
<td></td>
</tr>
<tr>
<td>open(path, *args, **kwargs)</td>
<td>Return a xarray.dataset from an url</td>
</tr>
<tr>
<td>open_dataset(url, *args, **kwargs)</td>
<td>Open multiple urls as a single xarray dataset.</td>
</tr>
<tr>
<td>read_csv(url, **kwargs)</td>
<td>Return a pandas.dataframe from an url that is a csv resource</td>
</tr>
<tr>
<td>register(uri)</td>
<td>Keep track of files open with this instance</td>
</tr>
<tr>
<td>store_path(uri)</td>
<td></td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protocol</td>
<td></td>
</tr>
</tbody>
</table>

argopy.stores.indexstore

class indexstore(cache: bool = False, cachedir: str = '', index_file: str = 'ar_index_global_prof.txt', **kw)

Use to manage access to a local Argo index and searches

__init__([cache: bool = False, cachedir: str = '', index_file: str = 'ar_index_global_prof.txt', **kw])

Create a file storage system for Argo index file requests

Parameters

- cache [bool (False)]
- cachedir [str (used value in global OPTIONS)]
- index_file: str (“ar_index_global_prof.txt”)
Methods

```python
__init__(cache, cachedir, index_file) Create a file storage system for Argo index file requests

cachepath(uri, errors) Return path to cached file for a given URI

clear_cache()

in_cache(fs, uri) Return true if uri is cached

in_memory(fs, uri) Return True if uri is in the memory store

open_index()

read_csv(search_cls) Run a search on an csv Argo index file and return a Pandas DataFrame with results

res2dataframe(results) Convert a csv like string into a DataFrame
```

argopy.stores.indexfilter_wmo

class indexfilter_wmo (WMO: list = [], CYC=None, **kwargs)

Index filter based on WMO and/or CYCLE_NUMER

This is intended to be used by instances of an indexstore

Examples

```python
# Create filters:
filt = index_filter_wmo(WMO=13857)  
filt = index_filter_wmo(WMO=13857, CYC=np.arange(1,10))  
filt = index_filter_wmo(WMO=[13857, 13858, 12], CYC=12)  
filt = index_filter_wmo(WMO=[13857, 13858, 12], CYC=[1, 12])  
filt = index_filter_wmo(CYC=250)  
filt = index_filter_wmo()

# Filter name: print(filt.uri())

# Direct usage:

with open("/Volumes/Data/ARGO/ar_index_global_prof.txt", "r") as f:
    results = filt.run(f)

# With the indexstore:

indexstore(cache=1, index_file="/Volumes/Data/ARGO/ar_index_global_prof.txt").open_dataframe(filt)

__init__(WMO: list = [], CYC=None, **kwargs)

Create Argo index filter for WMOs/CYCs

Parameters

WMO [list(int)] The list of WMOs to search

CYC [int, np.array(int), list(int)] The cycle numbers to search for each WMO

Methods

```python
__init__(WMO, CYC) Create Argo index filter for WMOs/CYCs

run(index_file) Run search on an Argo index file

uri() Return a unique name for this filter instance
```
Attributes

**sha**

Unique filter hash string

argopy.stores.indexfilter_box

class **indexfilter_box** *(BOX: list = [], **kwargs)*

Index filter based on LATITUDE, LONGITUDE, DATE

This is intended to be used by instances of an indexstore

Examples

# Create filters:

```python
filt = index_filter_box(BOX=[-70, -65, 30., 35.])
```

# Filter name:

```python
print(filt.uri())
```

# Direct usage:

```python
with open(‘/Volumes/Data/ARGO/ar_index_global_prof.txt’, ‘r’) as f:
results = filt.run(f)
```

# With the indexstore:

```python
indexstore(cache=1, index_file=’/Volumes/Data/ARGO/ar_index_global_prof.txt’).open_dataframe(filt)
```

__init__ *(BOX: list = [], **kwargs)*

Create Argo index filter for LATITUDE, LONGITUDE, DATE

Parameters

- **box** *(list(float, float, float, float, str, str))* The box domain to load all Argo data for: *box = [lon_min, lon_max, lat_min, lat_max, datim_min, datim_max]*

Methods

__init__ *(BOX)*

Create Argo index filter for LATITUDE, LONGITUDE, DATE

run(index_file)

Run search on an Argo index file

search_latlon(index, lon, lat)

Search

search_latlontim(index, lon, lat, tim)

Search

uri()

Return a unique name for this filter instance

Attributes

**sha**

Unique filter hash string
Fetchers

ERDDAP

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>argopy.data_fetchers.erddap_data.ErddapArgoDataFetcher(ds,...)</td>
<td>Manage access to Argo data through Ifremer ERDDAP</td>
</tr>
<tr>
<td>argopy.data_fetchers.erddap_data.Fetch_wmo(ds,...)</td>
<td>Manage access to Argo data through Ifremer ERDDAP for: a list of WMOs</td>
</tr>
<tr>
<td>argopy.data_fetchers.erddap_data.Fetch_box(ds,...)</td>
<td>Manage access to Argo data through Ifremer ERDDAP for: an ocean rectangle</td>
</tr>
</tbody>
</table>

argopy.data_fetchers.erddap_data.ErddapArgoDataFetcher


Managing access to Argo data through Ifremer ERDDAP

ERDDAP transaction are managed with the erddapy library


Instantiating 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.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>([ds, cache, cachedir, parallel, ...])</td>
<td>Instantiate an ERDDAP Argo data fetcher</td>
</tr>
<tr>
<td>clear_cache()</td>
<td>Remove cache files and entries from resources opened with this fetcher</td>
</tr>
<tr>
<td>cname()</td>
<td>Return a unique string defining the constraints</td>
</tr>
<tr>
<td>define_constraints()</td>
<td>Define erddapy constraints</td>
</tr>
<tr>
<td>filter_data_mode(ds, **kwargs)</td>
<td>Filter ERDDAP data based on constraints</td>
</tr>
<tr>
<td>filter_qc(ds, **kwargs)</td>
<td>Filter QC data based on constraints</td>
</tr>
<tr>
<td>filter_variables(ds[, mode])</td>
<td>Filter variables based on constraints</td>
</tr>
<tr>
<td>get_url()</td>
<td>Return the URL to download data requested</td>
</tr>
<tr>
<td>init(*args, **kwargs)</td>
<td>Initialisation for a specific fetcher</td>
</tr>
<tr>
<td>to_xarray([errors])</td>
<td>Load Argo data and return a xarray.DataSet</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_POINTS</td>
<td>Number of measurements expected to be returned by a request</td>
</tr>
<tr>
<td>cachepath</td>
<td>Return path to cached file(s) for this request</td>
</tr>
<tr>
<td>uri</td>
<td>Return the list of Unique Resource Identifier (URI) to download data</td>
</tr>
</tbody>
</table>

argopy.data_fetchers.erddap_data.Fetch_wmo


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(**)


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.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>([ds, cache, cachedir, parallel, ...])</td>
<td>Instantiate an ERDDAP Argo data fetcher</td>
</tr>
<tr>
<td>clear_cache()</td>
<td>Remove cache files and entries from resources opened with this fetcher</td>
</tr>
<tr>
<td>cname()</td>
<td>Return a unique string defining the constraints</td>
</tr>
<tr>
<td>dashboard(**kw)</td>
<td>Define erddap constraints</td>
</tr>
<tr>
<td>filter_data_mode(ds, **kwargs)</td>
<td>Return the URL to download data requested</td>
</tr>
<tr>
<td>filter_qc(ds, **kwargs)</td>
<td>Create Argo data loader for WMOs</td>
</tr>
<tr>
<td>filter_variables(ds[, model])</td>
<td>Load Argo data and return a xarray.DataSet</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_POINTS</td>
<td>Number of measurements expected to be returned by a request</td>
</tr>
<tr>
<td>cachepath</td>
<td>Return path to cached file(s) for this request</td>
</tr>
<tr>
<td>uri</td>
<td>List of URLs to load for a request</td>
</tr>
</tbody>
</table>

argopy.data_fetchers.erddap_data.Fetch_box

 Manage access to Argo data through Ifremer ERDDAP for: an ocean rectangle

 Instantiate an ERDDAP Argo data fetcher

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ds: str (optional)</td>
<td>Dataset to load: ‘phy’ or ‘ref’ or ‘bgc’</td>
</tr>
<tr>
<td>cache: bool (optional)</td>
<td>Cache data or not (default: False)</td>
</tr>
<tr>
<td>cachedir: str (optional)</td>
<td>Path to cache folder</td>
</tr>
</tbody>
</table>
parallel: bool (optional)  Chunk request to use parallel fetching (default: False)


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.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>Instantiate an ERDDAP Argo data fetcher</td>
</tr>
<tr>
<td>clear_cache()</td>
<td>Remove cache files and entries from resources opened with this fetcher</td>
</tr>
<tr>
<td>cname()</td>
<td>Return a unique string defining the constraints</td>
</tr>
<tr>
<td>define_constraints()</td>
<td>Define request constraints</td>
</tr>
<tr>
<td>filter_data_mode(ds, **kwargs)</td>
<td>Filter data mode</td>
</tr>
<tr>
<td>filter_qc(ds, **kwargs)</td>
<td>Filter quality control</td>
</tr>
<tr>
<td>filter_variables(ds[, mode])</td>
<td>Filter variables</td>
</tr>
<tr>
<td>get_url()</td>
<td>Return the URL to download data requested</td>
</tr>
<tr>
<td>init(box, **kw)</td>
<td>Create Argo data loader</td>
</tr>
<tr>
<td>to_xarray([errors])</td>
<td>Load Argo data and return a xarray.DataSet</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_POINTS</td>
<td>Number of measurements expected to be returned by a request</td>
</tr>
<tr>
<td>cachepath</td>
<td>Return path to cached file(s) for this request</td>
</tr>
<tr>
<td>uri</td>
<td>List of files to load for a request</td>
</tr>
</tbody>
</table>

Local FTP

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>argopy.data_fetchers.localftp_data.LocalFTPArgoDataFetcher(...)</td>
<td>Manage access to Argo data from a local copy of GDAC ftp</td>
</tr>
<tr>
<td>argopy.data_fetchers.localftp_data.Fetch_wmo(...)</td>
<td>Manage access to local ftp Argo data for: a list of WMOs</td>
</tr>
<tr>
<td>argopy.data_fetchers.localftp_data.Fetch_box(...)</td>
<td>Manage access to local ftp Argo data for: a rectangular space/time domain</td>
</tr>
</tbody>
</table>

Manage access to Argo data from a local copy of GDAC ftp


Init fetcher

Parameters

local_ftp: str (optional) Path to the local directory where the ‘dac’ folder is located.
ds: str (optional) Dataset to load: ‘phy’ or ‘ref’ or ‘bgc’
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.
cache: bool (optional) Cache data or not (default: False)
cachedir: str (optional) Path to cache folder
dimension: str 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.
parallel: bool (optional) Chunk request to use parallel fetching (default: False)
progress: bool (optional) Show a progress bar or not when fetching data.
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.
  • {'lon': 2} will create a maximum of 2 chunks along longitude when used with Fetch_box.
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.
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code> ([<code>local_ftp</code>, <code>ds</code>, <code>cache</code>, <code>cachedir</code>, ...])</td>
<td>Init fetcher</td>
</tr>
<tr>
<td><code>clear_cache()</code></td>
<td>Remove cache files and entries from resources opened with this fetcher</td>
</tr>
<tr>
<td><code>cname()</code></td>
<td>Return a unique string defining the constraints</td>
</tr>
<tr>
<td><code>filter_data_mode(ds, **kwargs)</code></td>
<td></td>
</tr>
<tr>
<td><code>filter_qc(ds, **kwargs)</code></td>
<td></td>
</tr>
<tr>
<td><code>filter_variables(ds[, mode])</code></td>
<td></td>
</tr>
<tr>
<td><code>get_path(wmo[, cyc])</code></td>
<td>Return the absolute path toward the netcdf source file of a given wmo/cyc pair and a dataset</td>
</tr>
<tr>
<td><code>init(*args, **kwargs)</code></td>
<td>Initialisation for a specific fetcher</td>
</tr>
<tr>
<td><code>to_xarray([errors])</code></td>
<td>Load Argo data and return a xarray.Dataset</td>
</tr>
</tbody>
</table>

## Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cachepath</code></td>
<td>Return path to cache file(s) for this request</td>
</tr>
<tr>
<td><code>uri</code></td>
<td>Return the list of files to load</td>
</tr>
</tbody>
</table>

### argopy.data_fetchers.localftp_data.Fetch_wmo

```python
```

Manage access to local ftp Argo data for: a list of WMOs

### __init__ ([`local_ftp`, `ds`, `cache`, `cachedir`, ...])

Init fetcher

#### Parameters

- **local_ftp**: `str` (optional) Path to the local directory where the ‘dac’ folder is located.
- **ds**: `str` (optional) Dataset to load: ‘phy’ or ‘ref’ or ‘bgc’
- **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.
- **cache**: `bool` (optional) Cache data or not (default: False)
- **cachedir**: `str` (optional) Path to cache folder
- **dimension**: `str` 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.
- **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.
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.
- {'lon': 2} will create a maximum of 2 chunks along longitude when used with Fetch_box.

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.

Methods

- **__init__**([local_ftp, ds, cache, cachedir, ...]) Init fetcher
- clear_cache() Remove cache files and entries from resources opened with this fetcher
- cname() Return a unique string defining the constraints
- dashboard(**kw) 
- filter_data_mode(ds,**kwargs) 
- filter_qc(ds,**kwargs) 
- filter_variables(ds[, mode])
- get_path(wmo[, cyc]) Return the absolute path toward the netcdf source file of a given wmo/cyc pair and a dataset
- init([WMO, CYC]) Create Argo data loader for WMOs
- to_xarray([errors]) Load Argo data and return a xarray.Dataset

Attributes

- cachepath Return path to cache file(s) for this request
- uri List of files to load for a request

argopy.data_fetchers.localftp_data.Fetch_box


Manage access to local ftp Argo data for: a rectangular space/time domain

- **__init__**([local_ftp = '', ds = '', cache = False, cachedir = '', dimension = 'point', errors = 'raise', parallel = False, parallel_method = 'thread', progress = False, chunks = 'auto', chunks_maxsize = dict = {}, **kwargs)) Init fetcher

Parameters

- local_ftp: str (optional) Path to the local directory where the ‘dac’ folder is located.
- ds: str (optional) Dataset to load: ‘phy’ or ‘ref’ or ‘bgc’
- 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.

**cache:** bool (optional) Cache data or not (default: False)

**cachedir:** str (optional) Path to cache folder

**dimension:** str 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.

**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.

**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.
- {’lon’: 2} will create a maximum of 2 chunks along longitude when used with Fetch_box.

**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.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong><strong>init</strong></strong></td>
<td>Init fetcher</td>
</tr>
<tr>
<td>clear_cache()</td>
<td>Remove cache files and entries from resources opened with this fetcher</td>
</tr>
<tr>
<td>cname()</td>
<td>Return a unique string defining the constraints</td>
</tr>
<tr>
<td>filter_data_mode(ds, **kwargs)</td>
<td></td>
</tr>
<tr>
<td>filter_qc(ds, **kwargs)</td>
<td></td>
</tr>
<tr>
<td>filter_variables(ds[, mode])</td>
<td></td>
</tr>
<tr>
<td>get_path(wmo[, cyc])</td>
<td>Return the absolute path toward the netcdf source file of a given wmo/cyc pair and a dataset</td>
</tr>
<tr>
<td>init(box)</td>
<td>Create Argo data loader</td>
</tr>
<tr>
<td>to_xarray([errors])</td>
<td>Load Argo data and return a xarray.Dataset</td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cachepath</td>
<td>Return path to cache file(s) for this request</td>
</tr>
<tr>
<td>url</td>
<td>List of files to load for a request</td>
</tr>
</tbody>
</table>
argopy Documentation, Release 999

Argovis

```
argopy.data_fetchers.argovis_data.
ArgovisDataFetcher(ds,...)
argopy.data_fetchers.argovis_data.
Fetch_wmo(ds,...)
argopy.data_fetchers.argovis_data.
Fetch_box(ds,...)
```

### argopy.data_fetchers.argovis_data.ArgovisDataFetcher

#### class ArgovisDataFetcher

```
```


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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>Instantiate an Argovis Argo data loader</td>
</tr>
<tr>
<td>clear_cache()</td>
<td>Remove cache files and entries from resources opened with this fetcher</td>
</tr>
<tr>
<td>cname()</td>
<td>Return a unique string defining the constraints</td>
</tr>
<tr>
<td>filter_data_mode(ds, **kwargs)</td>
<td>Enforce rectangular box shape</td>
</tr>
<tr>
<td>filter_domain(ds)</td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter_qc(ds, **kwargs)</td>
<td>Initialise for a specific fetcher</td>
</tr>
<tr>
<td>filter_variables(ds[, mode])</td>
<td>Convert json data to Pandas DataFrame</td>
</tr>
<tr>
<td>init()</td>
<td>Load Argo data and return a Pandas dataframe</td>
</tr>
<tr>
<td>json2dataframe(profiles)</td>
<td>Download and return data as xarray Datasets</td>
</tr>
</tbody>
</table>

**Attributes**

- cachepath: Return path to cache file for this request
- url: Return the URL used to download data

```python
class Fetch_wmo:
    def __init__(self, ds='', cache=False, cachedir='', parallel=False, parallel_method='thread',
                 progress=False, chunks='auto', chunks_maxsize={}, api_timeout=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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__(ds, cache, cachedir, parallel, ...)</code></td>
<td>Instantiate an Argovis Argo data loader</td>
</tr>
<tr>
<td><code>clear_cache()</code></td>
<td>Remove cache files and entries from resources opened with this fetcher</td>
</tr>
<tr>
<td><code>cname()</code></td>
<td>Return a unique string defining the constraints</td>
</tr>
<tr>
<td><code>dashboard(**kw)</code></td>
<td>Enforce rectangular box shape</td>
</tr>
<tr>
<td><code>filter_data_mode(ds, **kwargs)</code></td>
<td>Filter data mode</td>
</tr>
<tr>
<td><code>filter_domain(ds)</code></td>
<td>Filter domain</td>
</tr>
<tr>
<td><code>filter_qc(ds, **kwargs)</code></td>
<td>Filter quality control</td>
</tr>
<tr>
<td><code>filter_variables(ds[, mode])</code></td>
<td>Filter variables</td>
</tr>
<tr>
<td><code>get_url(wmo[, cyc])</code></td>
<td>Return path toward the source file of a given wmo/cyc pair</td>
</tr>
<tr>
<td><code>init([WMO, CYC])</code></td>
<td>Create Argo data loader for WMOs and CYCs</td>
</tr>
<tr>
<td><code>json2dataframe(profiles)</code></td>
<td>Convert json data to Pandas DataFrame</td>
</tr>
<tr>
<td><code>to_dataframe([errors])</code></td>
<td>Load Argo data and return a Pandas dataframe</td>
</tr>
<tr>
<td><code>to_xarray([errors])</code></td>
<td>Download and return data as Xarray Datasets</td>
</tr>
</tbody>
</table>

## Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cachepath</code></td>
<td>Return path to cache file for this request</td>
</tr>
<tr>
<td><code>uri</code></td>
<td>List of URLs to load for a request</td>
</tr>
</tbody>
</table>

---

**argopy.data_fetchers.argovis_data.Fetch_box**

```python
class Fetch_box:
    def __init__(ds='', cache=False, cachedir='', parallel=False, parallel_method='thread', progress=False, chunks='auto', chunks_maxsize={}, api_timeout=0, **kwargs):
        Instantiate an Argovis Argo data loader

Parameters

- `ds`: str (optional) Dataset to load: ‘phy’ or ‘bge’
- `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)
- `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

filter_data_mode(ds, **kwargs) Enforce rectangular box shape

filter_domain(ds) Enforce rectangular box shape

filter_qc(ds, **kwargs) Enforce rectangular box shape

filter_variables(ds[, mode]) Enforce rectangular box shape

gt_url() Return the URL used to download data

gt_url_rect() Return the URL used to download data

init(box) Create Argo data loader

json2dataframe(profiles) Convert json data to Pandas DataFrame

to_dataframe([errors]) Load Argo data and return a Pandas data frame

to_xarray([errors]) Download and return data as xarray Datasets

Attributes

cachepath Return path to cache file for this request

url List of URLs to load for a request

1.15.4 Xarray argo name space

class ArgoAccessor
Class registered under scope argo to access a xarray.Dataset object.

• Ensure all variables are of the Argo required dtype with:
  dsargo.cast_types()

• Convert a collection of points into a collection of profiles:
  dsargo.point2profile()

• Convert a collection of profiles to a collection of points:
  dsargo.profile2point()

cast_types() Make sure variables are of the appropriate types

This is hard coded, but should be retrieved from an API somewhere Should be able to handle all possible variables encountered in the Argo dataset
**filter_data_mode**

*keep_error: bool = True, errors: str = 'raise')*

Filter variables according to their data mode

This 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’)

**Parameters**

keep_error: bool, optional  If true (default) keep the measurements error fields or not.

effects: {'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.

**Returns**

xarray.Dataset

**filter_qc**

*(QC_list=[1, 2], drop=True, mode='all', mask=False)*

Filter data set according to QC values

Mask the dataset for 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.

**interp_std_levels**

*(std_lev)*

Returns a new dataset interpolated to new inputs levels

**Parameters**

list or np.array  Standard levels used for interpolation

**Returns**

xarray.Dataset

**point2profile**

Transform a collection of points into a collection of profiles

**profile2point**

Convert a collection of profiles to a collection of points

**teos10**

*(vlist: list = ['SA', 'CT', 'SIG0', 'N2', 'PV', 'PTEMP'], inplace: bool = True)*

Add TEOS10 variables to the dataset


**Parameters**

vlist: list(str)  List with the name of variables to add.

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

**Returns**

xarray.Dataset

**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)

Returns

int or tuple of int

Examples

unique_float_profile_id = uid(690024,13,'A')  # Encode wmo, cyc, drc = uid(unique_float_profile_id)  # Decode
[ADMT] See all the ADMT documentation here: http://www.argodatamgt.org/Documentation
a

argopy.xarray, 69
Symbols
__init__() (ArgovisDataFetcher method), 66
__init__() (ErddapArgoDataFetcher method), 58
__init__() (Fetch_box method), 60, 64, 68
__init__() (Fetch_wmo method), 59, 63, 67
__init__() (LocalFTPArgoDataFetcher method), 62
__init__() (filestore method), 52
__init__() (indexfilter_box method), 57
__init__() (indexfilter_wmo method), 56
__init__() (indexstore method), 55
__init__() (memorystore method), 54
__init__() (set_options method), 51

A
ArgoAccessor (class in argopy), 69
argopy.xarray module, 69
ArgovisDataFetcher (class in argopy.data_fetchers.argovis_data), 66

C
cast_types() (ArgoAccessor method), 69
clear_cache() (in module argopy), 51

D
dashboard() (in module argopy), 50
DataFetcher (in module argopy), 46

E
ErddapArgoDataFetcher (class in argopy.data_fetchers.erddap_data), 58

F
Fetch_box (class in argopy.data_fetchers.argovis_data), 68
Fetch_box (class in argopy.data_fetchers.erddap_data), 60
Fetch_box (class in argopy.data_fetchers.localftp_data), 64
Fetch_wmo (class in argopy.data_fetchers.argovis_data), 67
Fetch_wmo (class in argopy.data_fetchers.erddap_data), 59
Fetch_wmo (class in argopy.data_fetchers.localftp_data), 63
filestore (class in argopy.stores), 52
filter_data_mode() (ArgoAccessor method), 69
filter_qc() (ArgoAccessor method), 70
float() (DataFetcher method), 47
float() (IndexFetcher method), 48

H
httpstore (class in argopy.stores), 53

I
IndexFetcher (in module argopy), 47
indexfilter_box (class in argopy.stores), 57
indexfilter_wmo (class in argopy.stores), 56
indexstore (class in argopy.stores), 55
interp_std_levels() (ArgoAccessor method), 70

L
list_available_data_src() (in module argopy.utilities), 52
list_available_index_src() (in module argopy.utilities), 52
LocalFTPArgoDataFetcher (class in argopy.data_fetchers.localftp_data), 62

M
memorystore (class in argopy.stores), 54
module argopy.xarray, 69

O
open_dataset() (in module argopy.tutorial), 51

P
plot() (IndexFetcher method), 49
point2profile() (ArgoAccessor method), 70
profile() (DataFetcher method), 48
profile2point() (ArgoAccessor method), 70
| R         | region() (*DataFetcher method*), 47  |
|          | region() (*IndexFetcher method*), 48 |
| S         | set_options (*class in argopy*), 50  |
|           | show_versions() (*in module argopy*), 52 |
| T         | teos10() (*ArgoAccessor method*), 70  |
|           | to_csv() (*IndexFetcher method*), 49  |
|           | to_dataframe() (*DataFetcher method*), 49 |
|           | to_dataframe() (*IndexFetcher method*), 49 |
|           | to_xarray() (*DataFetcher method*), 48 |
|           | to_xarray() (*IndexFetcher method*), 49 |
| U         | uid() (*ArgoAccessor method*), 70  |
|           | uri() (*DataFetcher property*), 50  |