

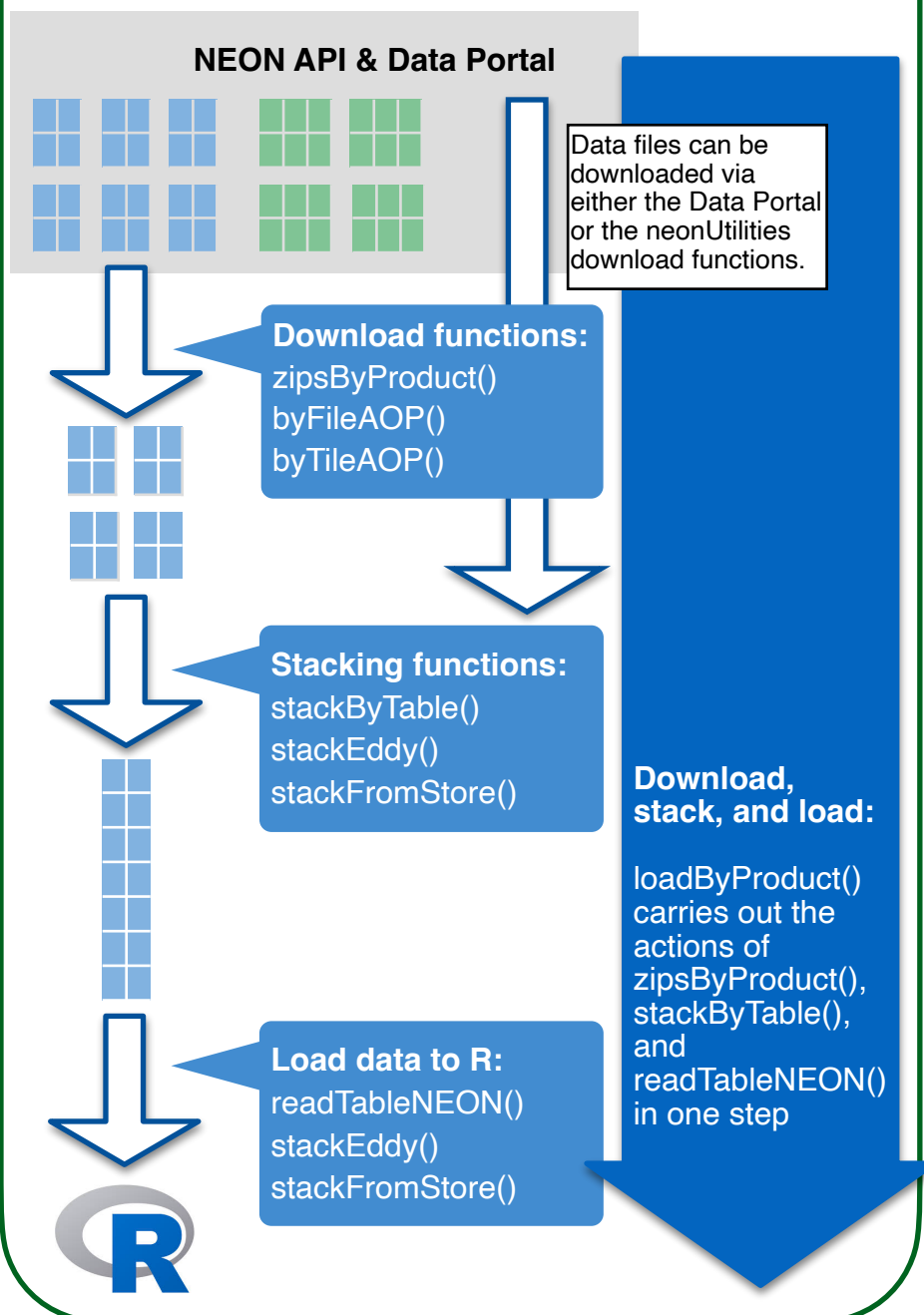
NEON data access and wrangling with neonUtilities

CHEAT SHEET Page 1

Overview

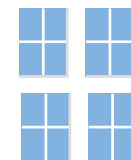
neonUtilities facilitates accessing and working with NEON data.

NEON data are published in discrete packages for each **data product**, **site**, and **month** of data collection. Each package may contain data from several **data tables** and/or **sensor locations**. **neonUtilities** can help you access those data packages, and also transform them to more tractable data formats.



Tabular data

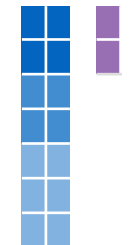
Tabular data published by NEON include both **observational** (human-collected, such as observations and measurements of birds and trees, and collection of physical samples) and **sensor** data. Data collected by sensors in the **surface-atmosphere exchange** system and the **remote sensing** platform are not tabular; see boxes below and to the right for these data.



NEON tabular data are provided in pre-packaged sets for each site and month with available data.

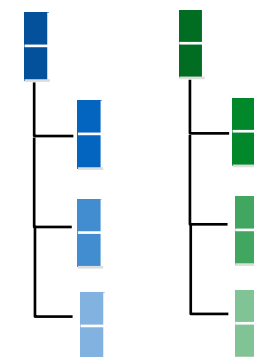


Within the site-month packages, data tables may contain the output of a single sensor, a specific field activity (observational data), or contextual data that are relevant to all time periods (sensor position files, and observational data collected once over the lifetime of the protocol, such as trap establishment or tree mapping data).



The stacking functions join the matching tables over each site and month, and over multiple sensors within a site, as necessary to create a single table for each data type. For contextual data that are relevant to all time periods, data are duplicated between site-month packages, so only the most recently published data are used in stacking.

Surface-atmosphere exchange



Surface-atmosphere exchange data are published in HDF5 files: Hierarchical Data Format, in which data are arranged in a systematic hierarchy, and descriptive metadata are available at each level in the hierarchy.



At the terminal nodes in this hierarchy, the data are generally tabular. `stackEddy()` extracts subsets of data from the terminal nodes and joins the matching variables over months and sensors, similar to the process for the sensor data stored in simple tabular format.

Remote sensing

`byFileAOP()` downloads all available data for a specified **data product**, **site**, and **year**. It can be used to download any remote sensing data product.

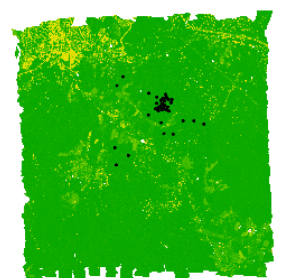
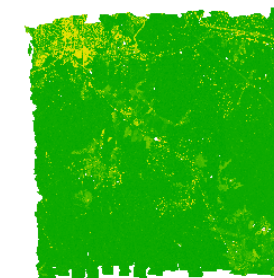
```
byFileAOP(dpID="DP3.30026.001",
  site="SCBI", year=2017)
```

SCBI:
Smithsonian
Conservation
Biology Institute

DP3.30026.001: Data
product ID for
Vegetation indices

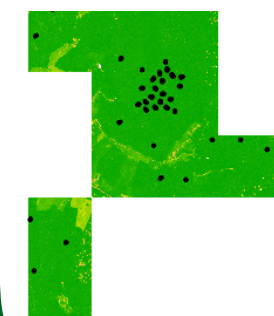
`byFileAOP()` downloads all available data. For example, here are the 143 tiles for the download specified above, which total 5 GB of data when uncompressed:

But downloading all 5 GB is excessive if you don't need it all. What if you're only interested in the tiles containing locations where trees have been measured on the ground, shown here in black dots? In that case, use `byTileAOP()` as described below.



```
byTileAOP(dpID="DP3.30026.001",
  site="SCBI", year=2017,
  easting=veg$easting,
  northing=veg$northing)
```

Easting and
northing vectors
obtained from
DP1.10098.001

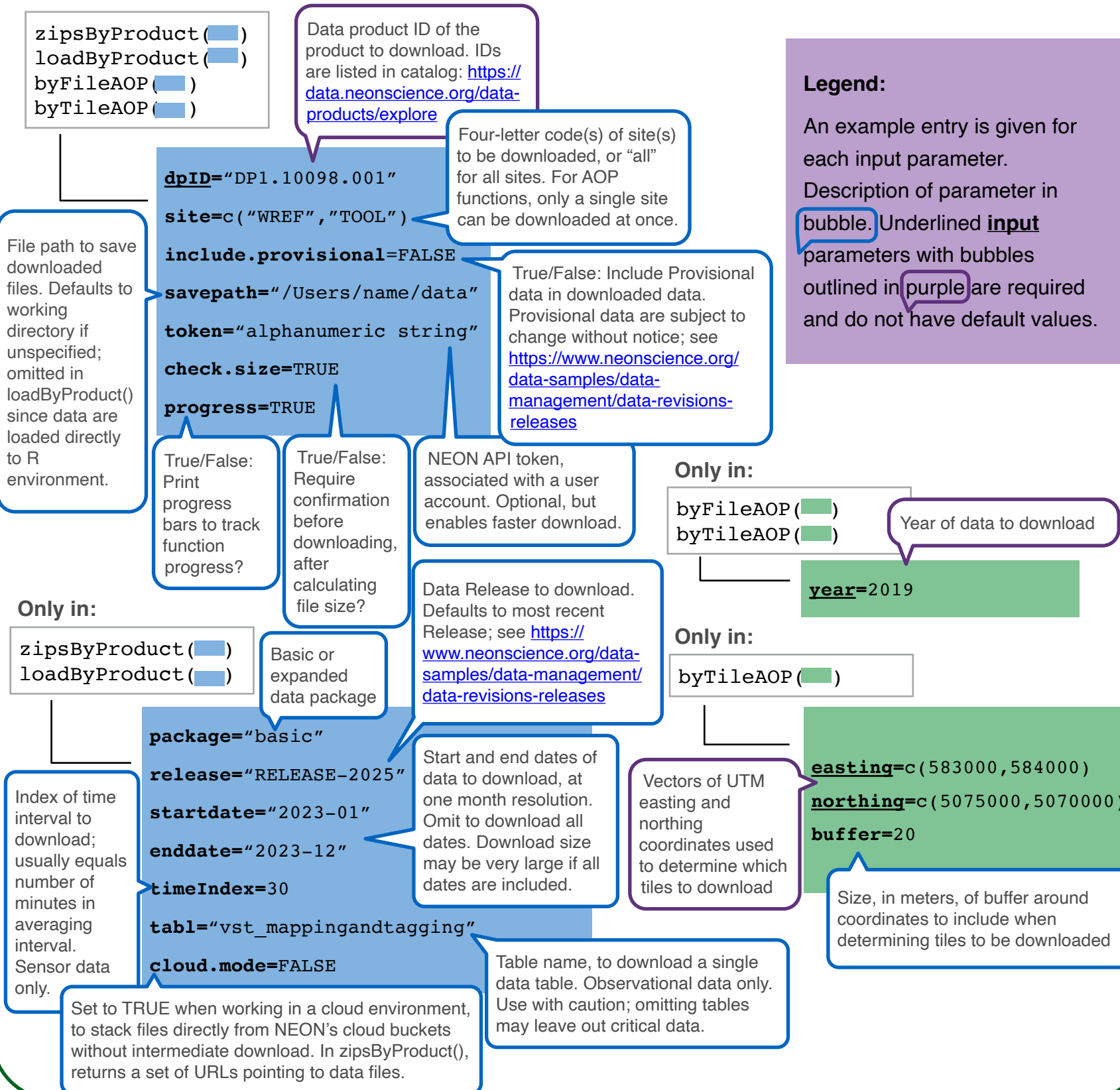


`byTileAOP()` downloads only the tiles corresponding to a specific set of coordinates, in this case only 10 out of 143 tiles! This option is only available for mosaicked (tiled) data products.

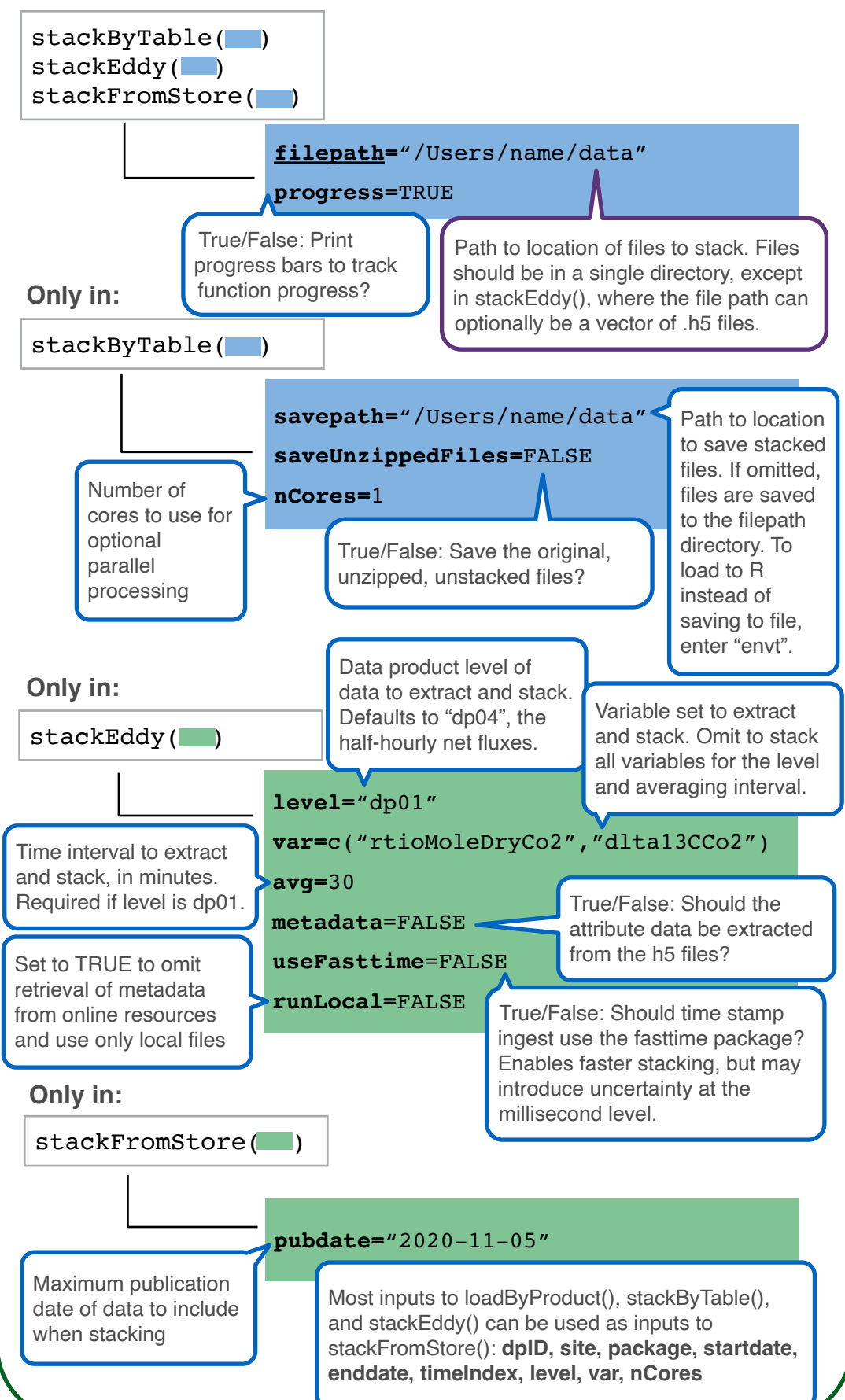
NEON data access and wrangling with neonUtilities

CHEAT SHEET Page 2

Function inputs: Download



Function inputs: Data stacking



NEON data access and wrangling with neonUtilities

CHEAT SHEET Page 3

Best practices

Avoid downloading the same data multiple times

loadByProduct() is the most popular choice for users downloading tabular data. But if you are writing a script you plan to run repeatedly, loadByProduct() will download the same data over and over, wasting a lot of time and compute power. Instead, use loadByProduct() to download the most recent data Release, save the data locally, and then use the local data files in your script.

For more details about data Releases and recommended workflows, see the tutorial [Understanding Releases and Provisional Data](#).

Manage download sizes

Many NEON data products are many gigabytes in size. Downloading and working with these data products can be challenging; they may overwhelm the compute capacity of the average computer. Suggestions for managing this challenge:

- Use byTileAOP() to download only the tiles of interest for remote sensing data
- Download remote sensing indices rather than full reflectance data sets, if the indices are sufficient for your analyses
- Download only the 30-minute data from sensor data products, if the 1-minute data aren't needed for your analyses
- For climatic data, use [Summary weather statistics \(DP4.00001.001\)](#)
- Consider using a cloud compute environment

Cite the data!

When you publish a manuscript using NEON data, cite the data used in your analyses. Citations are included in the outputs of neonUtilities data-stacking functions and remote sensing download functions, in BibTeX format. Citation text is also included on the Data Product Details page for each data product (for example, <https://data.neonscience.org/data-products/DP1.10026.001>).

For more information about best practices for citing NEON data, documentation, and resources, see [Acknowledging and Citing NEON](#).

Python

The Python package neonutilities (all lowercase!) brings neonUtilities functionality to Python. There are a few differences between the R and Python packages.

Function names and exceptions

R	Python
zipsByProduct()	zips_by_product()
stackByTable()	stack_by_table()
loadByProduct()	load_by_product()
byFileAOP()	by_file_aop()
byTileAOP()	by_tile_aop()
stackEddy()	stack_eddy()
stackFromStore()	Not included; neonstore integration is offered in R because neonstore is an R package

Function inputs

Inputs to the Python functions are largely the same as the R inputs, with a few syntax adjustments. For example, dots have functional meaning in Python, so cloud.mode= and include.provisional= are cloud_mode= and include_provisional= in Python.

The overall descriptions and function references on the first two pages of this cheat sheet apply to Python as well, with syntax changes as needed. For full function documentation in Python, see the [Read the Docs](#) page.

Function outputs

loadByProduct() in R returns a named list of tables; load_by_product() in Python returns a dictionary of tables. The names of tables and metadata objects match between the two languages.