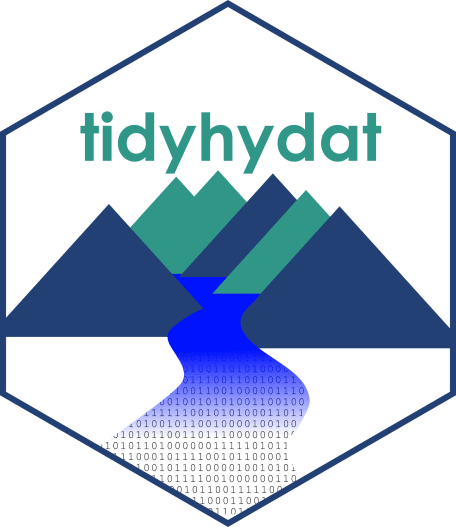
One of the best things about learning R is that no matter your skill level, there is always someone who can benefit from your experience. Topics in R ranging from complicated machine learning approaches to calculating a mean all find their relevant audiences. This is particularly true when writing R packages. With an ever evolving R package development landscape (R, GitHub, external data, CRAN, continuous integration, users), there is a strong possibility that you will be taken into regions of the R world that you never knew existed. More experienced developers may not get stuck in these regions and therefore not think to shine a light on them



tidyhydat is a new rOpenSci R package that provides a standard method of accessing [Environment and Climate Change Canada’s](https://www.canada.ca/en/environment-climate-change.html) [Water Survey of Canada](https://wateroffice.ec.gc.ca/index_e.html) data sources (WSC) using a consistent and easy to use interface. Many useRs will read this post and think “I’d know how to do that”. To you I tip my hat. To those who read this and think “huh, never knew that”, you are the target audience. In this post, I’ll focus on 5 things that I learned during the tidyhydat rOpenSci review, provide a brief introduction on how the package works and provide an example use case.

tidyhydat can be installed from CRAN by the usual manner:

install.packages("tidyhydat")

**Five things**

**1. Documentation**

A second and third set of eyes, in the form of rOpenSci reviewers, on the tidyhydat documentation added to the quality of the package immensely. It is so easy to write documentation that makes sense when you are writing the code, and makes zero sense to anyone else. Having reviewers review documentation and politely tap you on the shoulder to say “this is barely English” is the clear line between a package that is useful to only you and a package that is useful to others. From adding examples to each and every function, to suggestions that units be included in function documentation, all provide a more clear pathway for a potential new user to navigate the package.

**2. Where to store external data**

The single biggest feature of tidyhydat that caused the most headaches and learning opportunities for me is the fact that much of its capabilities rely on an external .sqlite3 data: HYDAT. HYDAT is the WSC’s quarterly published [archive of hydrometric data](http://collaboration.cmc.ec.gc.ca/cmc/hydrometrics/www/). Some stations data range extends into the 19^th^ century. It is, however, rather messy data and in a format (.sqlite3) that is likely unfamiliar to many useRs accustomed to flat data files like .csv, .txt or .xlsx.

tidyhydat seamlessly removes the need for a user to ever deal with the .sqlite3 file directly except running a function to download HYDAT upon installation:

download\_hydat()

Well, at least it does now. Previously, I had devised some convoluted solution that involved downloading the data to a specific directory then setting that directory in your .Renviron file. That is when one of the reviewers suggested the wonderfully simple rappirs package. Here is the description of the rappdirs package:

An easy way to determine which directories on the users computer you should use to save data, caches and logs.

This is exactly what I needed. Via rappdirs::user\_data\_dir() I was able to specify a directory to download the database that is OS-independent then tell every hy\_\* function to look for HYDAT in that directory.

**3. Prefix-style function naming**

In addition to interacting with HYDAT, tidyhydat also provides a series of functions to import real-time hydrometric data into R. Initially, to distinguish real-time functions from those that access HYDAT, I employed a confusing scheme where HYDAT functions were capitalized (DLY\_FLOWS) and real-time data functions were presented as lowercase (download\_realtime). Luckily another helpful suggestion by a reviewer was to employ consistent prefix-style naming rather than capitalization to accomplish this. Along with a slight function name change, DLY\_FLOWS() became hy\_daily\_flows(). This small change actually solves several problems at once:

* Prefixes act as sort of a package specific namespace helping to avoid any conflicts with other packages.
* Prefixes facilitate auto-completion because users only have to type hy\_ (+ TAB) at which point they can choose the function they would like to use.
* Enable a user to immediately know if a function accesses HYDAT. This unified interface provides clarity and consistency.

**4. Vignettes that use external data on CRAN**

One reviewer suggested that a vignette which worked through a complete analysis may be useful to a new user. This was a great idea. One problem: Since the package was destined for CRAN it needed to meet all the associated requirements. With respect to long-form documentation CRAN does not build vignettes and only makes sure they are actually runnable; it uses what you’ve built locally. To run through an example analysis, I needed to access the HYDAT database which is not shipped to CRAN.

**5. An exercise in reproducible science.**

tidyhydat was originally developed to save time accessing data. Instead of frequently opening database connections, making approximately the same query, forgetting to close the database connection, trying to reopen it, cursing R and generally blaming all the wrong people, I developed a package. However, after using the package it has become clear that it has a much deeper and useful feature — it is a tool that facilitates reproducible science and can provide reproducible workflows that support science-based decisions around water flow. It is now possible to set up code-based workflows that are re-runnable with more current data, different stations or regions over a range of time scales. The importance of this cannot be overstated and will be the focus of the remainder of this post.

**A reproducible work flow**

To illustrate a reproducible workflow facilitated by tidyhydat and made possible by the capabilities of R, we can ask a simple question related to a hydrometric station: how unique is this station compared to others in the same watershed? In this example we will consider the FRASER RIVER AT MISSION station. Records extend over many years and we can compare an overlapping record using correlation to develop a measure of **uniqueness** for this station. But first we need to engage in some preamble:

**Required packages**

A portion of the package audience is likely those with little or no experience in R so an example of the type of analysis possible with tidyhydat is helpful. For our purposes here, we need to install a whole other slew of packages. The packages needed for this analysis mostly fall into either the tidyverse collection of packages or the igraph family of packages. Several of these packages (dplyr, tidyr) were already installed alongside tidyhydat. The remaining packages can be installed using:

install.packages(c("purrr","corrr","igraph","ggraph","ggmap"))

then load the required packages:

library(tidyhydat)

library(dplyr)

library(tidyr)

library(corrr)

library(igraph)

library(ggraph)

library(ggmap)

**Using tidyhydat**

The first step when using tidyhydat will often start with the hy\_stations() function which holds metadata for all stations (active or discontinued) in HYDAT. WSC has a standard 7 digit station number format which contains hierarchical watershed information. The FRASER RIVER AT MISSION (08MH024) is located in the sub-sub-drainage Lower Fraser – Nahatlatch (08MH), in the sub-drainage Lower Fraser (08M) in the Pacific Ocean drainage (08). We can leverage that embedded watershed information and generate a list of station ID’s for all stations within our example watershed the Lower Fraser – Nahatlatch (08MH). An English description of the code below would follow as “Extract all stations from HYDAT then extract the first four digits from STATION\_NUMBER then filter for some sub-sub-drainage, then filter the stations for those that are active, then extract the station number as a vector”:

daily\_flows\_boxed <- hy\_stations() %>%

mutate(SUB\_SUB\_DRAINAGE\_AREA\_CD = substr(STATION\_NUMBER, 1,4)) %>%

filter(SUB\_SUB\_DRAINAGE\_AREA\_CD == "08MH") %>%

filter(HYD\_STATUS == "ACTIVE") %>%

pull(STATION\_NUMBER)

daily\_flows\_boxed

## [1] "08MH001" "08MH002" "08MH005" "08MH006" "08MH016" "08MH024" "08MH029"

## [8] "08MH035" "08MH056" "08MH076" "08MH090" "08MH098" "08MH103" "08MH126"

## [15] "08MH141" "08MH147" "08MH149" "08MH152" "08MH153" "08MH155" "08MH156"

## [22] "08MH166" "08MH167" "08MH168"

To move from the station vector to daily flow information we simply add another tidyhydat function hy\_daily\_flows() to the pipe:

daily\_flows\_boxed <- hy\_stations() %>%

mutate(SUB\_SUB\_DRAINAGE\_AREA\_CD = substr(STATION\_NUMBER, 1,4)) %>%

filter(SUB\_SUB\_DRAINAGE\_AREA\_CD == "08MH") %>%

filter(HYD\_STATUS == "ACTIVE") %>%

pull(STATION\_NUMBER) %>%

hy\_daily\_flows()

daily\_flows\_boxed

## # A tibble: 320,307 x 5

## STATION\_NUMBER Date Parameter Value Symbol

##

## 1 08MH005 1911-10-01 Flow NA

## 2 08MH006 1911-10-01 Flow NA

## 3 08MH005 1911-10-02 Flow NA

## 4 08MH006 1911-10-02 Flow NA

## 5 08MH005 1911-10-03 Flow NA

## 6 08MH006 1911-10-03 Flow NA

## 7 08MH005 1911-10-04 Flow NA

## 8 08MH006 1911-10-04 Flow NA

## 9 08MH005 1911-10-05 Flow NA

## 10 08MH006 1911-10-05 Flow NA

## # ... with 320,297 more rows

Now that we have all our hydrometric data tidy and in R, I’ll outline a cool example of what you can do with it.

**Correlate all stations in Fraser River – Lower Nahatlatch (08MH)**

Remember, we are trying to evaluate the uniqueness of the FRASER RIVER AT MISSION via correlation as compared to other stations in the watershed. We can utilize the excellent corrr package, specifically correlate() and stretch(), to calculate the correlation between each station all in a pipe. This series of piped functions takes all the daily flow data (daily\_flows\_boxed), converts it to long format, removes the columns we don’t want, correlates each column, stretches the data into long form and gets rid of correlations that results in NA’s:

Library(corrr)

cor\_df <- daily\_flows\_boxed %>%

spread(STATION\_NUMBER, Value) %>%

select(-Date, -Symbol, -Parameter) %>%

correlate() %>%

stretch() %>%

filter(!is.na(r))

**igraph munging**

At this point we need to begin using the igraph package for network analysis. Our first step is to make a graph for all the correlations then construct a subgraph for only the 08MH024 station:

Library(igraph)

## Convert to an igraph object for plotting

graph\_correlation <- graph\_from\_data\_frame(cor\_df)

## Subset for those edges from 08MH024

graph\_sub <- subgraph.edges(graph\_correlation, E(graph\_correlation)[inc('08MH024')])

**Create plot**

Finally, our last step is to plot each point in a spatially accurate manner and apply the results from the igraph object. The plotting step graphs the station location against satellite imagery using the ggmap and ggraph packages and connect stations with coloured lines that reflect the strength of the correlation:

Library(ggmap)

Library(ggraph)

## Get the lat long data, isolate the coordinates and rename to x and y

latlong\_layout\_ggmap <- hy\_stations(station\_number = V(graph\_sub)$name) %>%

select(LONGITUDE, LATITUDE) %>%

rename(x = LONGITUDE, y = LATITUDE)

## Create the spatial layout based on latitude and longitude.

spatial\_layout\_ggmap <- create\_layout(graph = graph\_sub,

layout = "manual",

node.positions = latlong\_layout\_ggmap)

## Acquire the static map

map <- get\_map(c(min(latlong\_layout\_ggmap$x),

min(latlong\_layout\_ggmap$y),

max(latlong\_layout\_ggmap$x),

max(latlong\_layout\_ggmap$y)),

source = "google", maptype = 'satellite', zoom = 9)

## Plot the igraph object

ggmap(map, base\_layer = ggraph(spatial\_layout\_ggmap), device = "extent") +

geom\_edge\_link(aes(color = r), edge\_width = 1) +

guides(edge\_alpha = "none") +

scale\_edge\_colour\_gradientn(name = "Correlation \nCoefficient", colours = rainbow(8)) +

geom\_node\_point(colour = "white", size = 4) +

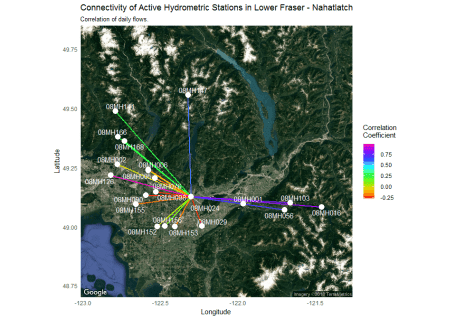
geom\_node\_text(aes(label = name), repel = TRUE, colour = "white") +

theme\_minimal() +

labs(y = "Latitude", x = "Longitude",

title = "Connectivity of Active Hydrometric Stations in Lower Fraser - Nahatlatch",

subtitle = "Correlation of daily flows.")



**What does the plot tell us**

Our original objective was to assess if the 08MH024 station was a **unique** station. This type of plot provides a clear pattern of hydrologic linkages. Flows at 08MH024 are more similar to stations at higher elevations (e.g. 08MH016) and less similar to lower elevation stations (e.g. 08MH002) demonstrating a spatial pattern of uniqueness. This likely reflects the importance of upstream hydrologic processes. The Fraser River, the largest river in British Columbia drains a huge area (228000 square km) where the flows in the majority of contributing upstream tributaries are driven by snow melt. This station, however, is located near the coastal environment where river flow is driven more by rainfall. This helps us connect upstream and downstream processes and evaluates the relative contributions to flow in the main river. A visual tool, such as the one illustrated here, provides a clear example of using the capabilities of R after bringing in a rich hydrometric data set, and highlights a reproducible hydrological workflow in R.

Tidy Data Principles

**Data tidying**

It is often said that 80% of data analysis is spent on the cleaning and preparing data. And it’s not just a first step, but it must be repeated many times over the course of analysis as new problems come to light or new data is collected. To get a handle on the problem, this paper focuses on a small, but important, aspect of data cleaning that I call data **tidying**: structuring datasets to facilitate analysis.

The principles of tidy data provide a standard way to organise data values within a dataset. A standard makes initial data cleaning easier because you don’t need to start from scratch and reinvent the wheel every time. The tidy data standard has been designed to facilitate initial exploration and analysis of the data, and to simplify the development of data analysis tools that work well together. Current tools often require translation. You have to spend time munging the output from one tool so you can input it into another. Tidy datasets and tidy tools work hand in hand to make data analysis easier, allowing you to focus on the interesting domain problem, not on the uninteresting logistics of data.

**Defining tidy data**

Happy families are all alike; every unhappy family is unhappy in its own way — Leo Tolstoy

Like families, tidy datasets are all alike but every messy dataset is messy in its own way. Tidy datasets provide a standardized way to link the structure of a dataset (its physical layout) with its semantics (its meaning). In this section, I’ll provide some standard vocabulary for describing the structure and semantics of a dataset, and then use those definitions to define tidy data.

**Data structure**

Most statistical datasets are data frames made up of **rows** and **columns**. The columns are almost always labeled and the rows are sometimes labeled. The following code provides some data about an imaginary classroom in a format commonly seen in the wild. The table has three columns and four rows, and both rows and columns are labeled.

classroom <- read.csv("classroom.csv", stringsAsFactors = FALSE)

classroom

*#> name quiz1 quiz2 test1*

*#> 1 Billy <NA> D C*

*#> 2 Suzy F <NA> <NA>*

*#> 3 Lionel B C B*

*#> 4 Jenny A A B*

There are many ways to structure the same underlying data. The following table shows the same data as above, but the rows and columns have been transposed.

read.csv("classroom2.csv", stringsAsFactors = FALSE)

*#> assessment Billy Suzy Lionel Jenny*

*#> 1 quiz1 <NA> FALSE B A*

*#> 2 quiz2 D NA C A*

*#> 3 test1 C NA B B*

The data is the same, but the layout is different. Our vocabulary of rows and columns is simply not rich enough to describe why the two tables represent the same data. In addition to appearance, we need a way to describe the underlying semantics, or meaning, of the values displayed in the table.

**Data semantics**

A dataset is a collection of **values**, usually either numbers (if quantitative) or strings (if qualitative). Values are organised in two ways. Every value belongs to a **variable** and an **observation**. A variable contains all values that measure the same underlying attribute (like height, temperature, duration) across units. An observation contains all values measured on the same unit (like a person, or a day, or a race) across attributes.

A tidy version of the classroom data looks like this: (you’ll learn how the functions work a little later)

library(tidyr)

library(dplyr)

classroom2 <- classroom %>%

pivot\_longer(quiz1:test1, names\_to = "assessment", values\_to = "grade") %>%

arrange(name, assessment)

classroom2

*#> # A tibble: 12 × 3*

*#> name assessment grade*

*#> <chr> <chr> <chr>*

*#> 1 Billy quiz1 <NA>*

*#> 2 Billy quiz2 D*

*#> 3 Billy test1 C*

*#> 4 Jenny quiz1 A*

*#> 5 Jenny quiz2 A*

*#> 6 Jenny test1 B*

*#> # … with 6 more rows*

This makes the values, variables, and observations more clear. The dataset contains 36 values representing three variables and 12 observations. The variables are:

1. name, with four possible values (Billy, Suzy, Lionel, and Jenny).
2. assessment, with three possible values (quiz1, quiz2, and test1).
3. grade, with five or six values depending on how you think of the missing value (A, B, C, D, F, NA).

The tidy data frame explicitly tells us the definition of an observation. In this classroom, every combination of name and assessment is a single measured observation. The dataset also informs us of missing values, which can and do have meaning. Billy was absent for the first quiz, but tried to salvage his grade. Suzy failed the first quiz, so she decided to drop the class. To calculate Billy’s final grade, we might replace this missing value with an F (or he might get a second chance to take the quiz). However, if we want to know the class average for Test 1, dropping Suzy’s structural missing value would be more appropriate than imputing a new value.

For a given dataset, it’s usually easy to figure out what are observations and what are variables, but it is surprisingly difficult to precisely define variables and observations in general. For example, if the columns in the classroom data were height and weight we would have been happy to call them variables. If the columns were height and width, it would be less clear cut, as we might think of height and width as values of a dimension variable. If the columns were home phone and work phone, we could treat these as two variables, but in a fraud detection environment we might want variables phone number and number type because the use of one phone number for multiple people might suggest fraud. A general rule of thumb is that it is easier to describe functional relationships between variables (e.g., z is a linear combination of x and y, density is the ratio of weight to volume) than between rows, and it is easier to make comparisons between groups of observations (e.g., average of group a vs. average of group b) than between groups of columns.

In a given analysis, there may be multiple levels of observation. For example, in a trial of new allergy medication we might have three observational types: demographic data collected from each person (age, sex, race), medical data collected from each person on each day (number of sneezes, redness of eyes), and meteorological data collected on each day (temperature, pollen count).

Variables may change over the course of analysis. Often the variables in the raw data are very fine grained, and may add extra modelling complexity for little explanatory gain. For example, many surveys ask variations on the same question to better get at an underlying trait. In early stages of analysis, variables correspond to questions. In later stages, you change focus to traits, computed by averaging together multiple questions. This considerably simplifies analysis because you don’t need a hierarchical model, and you can often pretend that the data is continuous, not discrete.

**Tidy data**

Tidy data is a standard way of mapping the meaning of a dataset to its structure. A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types. In **tidy data**:

1. Every column is a variable.
2. Every row is an observation.
3. Every cell is a single value.

This is Codd’s 3rd normal form, but with the constraints framed in statistical language, and the focus put on a single dataset rather than the many connected datasets common in relational databases. **Messy data** is any other arrangement of the data.

Tidy data makes it easy for an analyst or a computer to extract needed variables because it provides a standard way of structuring a dataset. Compare the different versions of the classroom data: in the messy version you need to use different strategies to extract different variables. This slows analysis and invites errors. If you consider how many data analysis operations involve all of the values in a variable (every aggregation function), you can see how important it is to extract these values in a simple, standard way. Tidy data is particularly well suited for vectorised programming languages like R, because the layout ensures that values of different variables from the same observation are always paired.

While the order of variables and observations does not affect analysis, a good ordering makes it easier to scan the raw values. One way of organising variables is by their role in the analysis: are values fixed by the design of the data collection, or are they measured during the course of the experiment? Fixed variables describe the experimental design and are known in advance. Computer scientists often call fixed variables dimensions, and statisticians usually denote them with subscripts on random variables. Measured variables are what we actually measure in the study. Fixed variables should come first, followed by measured variables, each ordered so that related variables are contiguous. Rows can then be ordered by the first variable, breaking ties with the second and subsequent (fixed) variables. This is the convention adopted by all tabular displays in this paper.

**Tidying messy datasets**

Real datasets can, and often do, violate the three precepts of tidy data in almost every way imaginable. While occasionally you do get a dataset that you can start analysing immediately, this is the exception, not the rule. This section describes the five most common problems with messy datasets, along with their remedies:

* Column headers are values, not variable names.
* Multiple variables are stored in one column.
* Variables are stored in both rows and columns.
* Multiple types of observational units are stored in the same table.
* A single observational unit is stored in multiple tables.

Surprisingly, most messy datasets, including types of messiness not explicitly described above, can be tidied with a small set of tools: pivoting (longer and wider) and separating. The following sections illustrate each problem with a real dataset that I have encountered, and show how to tidy them.

**Column headers are values, not variable names**

A common type of messy dataset is tabular data designed for presentation, where variables form both the rows and columns, and column headers are values, not variable names. While I would call this arrangement messy, in some cases it can be extremely useful. It provides efficient storage for completely crossed designs, and it can lead to extremely efficient computation if desired operations can be expressed as matrix operations.

The following code shows a subset of a typical dataset of this form. This dataset explores the relationship between income and religion in the US. It comes from a report produced by the Pew Research Center, an American think-tank that collects data on attitudes to topics ranging from religion to the internet, and produces many reports that contain datasets in this format.

relig\_income

*#> # A tibble: 18 × 11*

*#> religion `<$10k` `$10-20k` `$20-30k` `$30-40k` `$40-50k` `$50-75k` `$75-100k`*

*#> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>*

*#> 1 Agnostic 27 34 60 81 76 137 122*

*#> 2 Atheist 12 27 37 52 35 70 73*

*#> 3 Buddhist 27 21 30 34 33 58 62*

*#> 4 Catholic 418 617 732 670 638 1116 949*

*#> 5 Don’t kn… 15 14 15 11 10 35 21*

*#> 6 Evangeli… 575 869 1064 982 881 1486 949*

*#> # … with 12 more rows, and 3 more variables: `$100-150k` <dbl>, `>150k` <dbl>,*

*#> # `Don't know/refused` <dbl>*

This dataset has three variables, religion, income and frequency. To tidy it, we need to **pivot** the non-variable columns into a two-column key-value pair. This action is often described as making a wide dataset longer (or taller).

When pivoting variables, we need to provide the name of the new key-value columns to create. After defining the colums to pivot (every column except for religion), you will need the name of the key column, which is the name of the variable defined by the values of the column headings. In this case, it’s income. The second argument is the name of the value column, frequency.

relig\_income %>%

pivot\_longer(-religion, names\_to = "income", values\_to = "frequency")

*#> # A tibble: 180 × 3*

*#> religion income frequency*

*#> <chr> <chr> <dbl>*

*#> 1 Agnostic <$10k 27*

*#> 2 Agnostic $10-20k 34*

*#> 3 Agnostic $20-30k 60*

*#> 4 Agnostic $30-40k 81*

*#> 5 Agnostic $40-50k 76*

*#> 6 Agnostic $50-75k 137*

*#> # … with 174 more rows*

This form is tidy because each column represents a variable and each row represents an observation, in this case a demographic unit corresponding to a combination of religion and income.

This format is also used to record regularly spaced observations over time. For example, the Billboard dataset shown below records the date a song first entered the billboard top 100. It has variables for artist, track, date.entered, rank and week. The rank in each week after it enters the top 100 is recorded in 75 columns, wk1 to wk75. This form of storage is not tidy, but it is useful for data entry. It reduces duplication since otherwise each song in each week would need its own row, and song metadata like title and artist would need to be repeated. This will be discussed in more depth in [multiple types](https://cran.r-project.org/web/packages/tidyr/vignettes/tidy-data.html#multiple-types).

billboard

*#> # A tibble: 317 × 79*

*#> artist track date.entered wk1 wk2 wk3 wk4 wk5 wk6 wk7 wk8*

*#> <chr> <chr> <date> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>*

*#> 1 2 Pac Baby… 2000-02-26 87 82 72 77 87 94 99 NA*

*#> 2 2Ge+her The … 2000-09-02 91 87 92 NA NA NA NA NA*

*#> 3 3 Doors Do… Kryp… 2000-04-08 81 70 68 67 66 57 54 53*

*#> 4 3 Doors Do… Loser 2000-10-21 76 76 72 69 67 65 55 59*

*#> 5 504 Boyz Wobb… 2000-04-15 57 34 25 17 17 31 36 49*

*#> 6 98^0 Give… 2000-08-19 51 39 34 26 26 19 2 2*

*#> # … with 311 more rows, and 68 more variables: wk9 <dbl>, wk10 <dbl>,*

*#> # wk11 <dbl>, wk12 <dbl>, wk13 <dbl>, wk14 <dbl>, wk15 <dbl>, wk16 <dbl>,*

*#> # wk17 <dbl>, wk18 <dbl>, wk19 <dbl>, wk20 <dbl>, wk21 <dbl>, wk22 <dbl>,*

*#> # wk23 <dbl>, wk24 <dbl>, wk25 <dbl>, wk26 <dbl>, wk27 <dbl>, wk28 <dbl>,*

*#> # wk29 <dbl>, wk30 <dbl>, wk31 <dbl>, wk32 <dbl>, wk33 <dbl>, wk34 <dbl>,*

*#> # wk35 <dbl>, wk36 <dbl>, wk37 <dbl>, wk38 <dbl>, wk39 <dbl>, wk40 <dbl>,*

*#> # wk41 <dbl>, wk42 <dbl>, wk43 <dbl>, wk44 <dbl>, wk45 <dbl>, wk46 <dbl>, …*

To tidy this dataset, we first use pivot\_longer() to make the dataset longer. We transform the columns from wk1 to wk76, making a new column for their names, week, and a new value for their values, rank:

billboard2 <- billboard %>%

pivot\_longer(

wk1:wk76,

names\_to = "week",

values\_to = "rank",

values\_drop\_na = TRUE

)

billboard2

*#> # A tibble: 5,307 × 5*

*#> artist track date.entered week rank*

*#> <chr> <chr> <date> <chr> <dbl>*

*#> 1 2 Pac Baby Don't Cry (Keep... 2000-02-26 wk1 87*

*#> 2 2 Pac Baby Don't Cry (Keep... 2000-02-26 wk2 82*

*#> 3 2 Pac Baby Don't Cry (Keep... 2000-02-26 wk3 72*

*#> 4 2 Pac Baby Don't Cry (Keep... 2000-02-26 wk4 77*

*#> 5 2 Pac Baby Don't Cry (Keep... 2000-02-26 wk5 87*

*#> 6 2 Pac Baby Don't Cry (Keep... 2000-02-26 wk6 94*

*#> # … with 5,301 more rows*

Here we use values\_drop\_na = TRUE to drop any missing values from the rank column. In this data, missing values represent weeks that the song wasn’t in the charts, so can be safely dropped.

In this case it’s also nice to do a little cleaning, converting the week variable to a number, and figuring out the date corresponding to each week on the charts:

billboard3 <- billboard2 %>%

mutate(

week = as.integer(gsub("wk", "", week)),

date = as.Date(date.entered) + 7 \* (week - 1),

date.entered = NULL

)

billboard3

*#> # A tibble: 5,307 × 5*

*#> artist track week rank date*

*#> <chr> <chr> <int> <dbl> <date>*

*#> 1 2 Pac Baby Don't Cry (Keep... 1 87 2000-02-26*

*#> 2 2 Pac Baby Don't Cry (Keep... 2 82 2000-03-04*

*#> 3 2 Pac Baby Don't Cry (Keep... 3 72 2000-03-11*

*#> 4 2 Pac Baby Don't Cry (Keep... 4 77 2000-03-18*

*#> 5 2 Pac Baby Don't Cry (Keep... 5 87 2000-03-25*

*#> 6 2 Pac Baby Don't Cry (Keep... 6 94 2000-04-01*

*#> # … with 5,301 more rows*

Finally, it’s always a good idea to sort the data. We could do it by artist, track and week:

billboard3 %>% arrange(artist, track, week)

*#> # A tibble: 5,307 × 5*

*#> artist track week rank date*

*#> <chr> <chr> <int> <dbl> <date>*

*#> 1 2 Pac Baby Don't Cry (Keep... 1 87 2000-02-26*

*#> 2 2 Pac Baby Don't Cry (Keep... 2 82 2000-03-04*

*#> 3 2 Pac Baby Don't Cry (Keep... 3 72 2000-03-11*

*#> 4 2 Pac Baby Don't Cry (Keep... 4 77 2000-03-18*

*#> 5 2 Pac Baby Don't Cry (Keep... 5 87 2000-03-25*

*#> 6 2 Pac Baby Don't Cry (Keep... 6 94 2000-04-01*

*#> # … with 5,301 more rows*

Or by date and rank:

billboard3 %>% arrange(date, rank)

*#> # A tibble: 5,307 × 5*

*#> artist track week rank date*

*#> <chr> <chr> <int> <dbl> <date>*

*#> 1 Lonestar Amazed 1 81 1999-06-05*

*#> 2 Lonestar Amazed 2 54 1999-06-12*

*#> 3 Lonestar Amazed 3 44 1999-06-19*

*#> 4 Lonestar Amazed 4 39 1999-06-26*

*#> 5 Lonestar Amazed 5 38 1999-07-03*

*#> 6 Lonestar Amazed 6 33 1999-07-10*

*#> # … with 5,301 more rows*

**Multiple variables stored in one column**

After pivoting columns, the key column is sometimes a combination of multiple underlying variable names. This happens in the tb (tuberculosis) dataset, shown below. This dataset comes from the World Health Organisation, and records the counts of confirmed tuberculosis cases by country, year, and demographic group. The demographic groups are broken down by sex (m, f) and age (0-14, 15-25, 25-34, 35-44, 45-54, 55-64, unknown).

tb <- as\_tibble(read.csv("tb.csv", stringsAsFactors = FALSE))

tb

*#> # A tibble: 5,769 × 22*

*#> iso2 year m04 m514 m014 m1524 m2534 m3544 m4554 m5564 m65 mu f04*

*#> <chr> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int>*

*#> 1 AD 1989 NA NA NA NA NA NA NA NA NA NA NA*

*#> 2 AD 1990 NA NA NA NA NA NA NA NA NA NA NA*

*#> 3 AD 1991 NA NA NA NA NA NA NA NA NA NA NA*

*#> 4 AD 1992 NA NA NA NA NA NA NA NA NA NA NA*

*#> 5 AD 1993 NA NA NA NA NA NA NA NA NA NA NA*

*#> 6 AD 1994 NA NA NA NA NA NA NA NA NA NA NA*

*#> # … with 5,763 more rows, and 9 more variables: f514 <int>, f014 <int>,*

*#> # f1524 <int>, f2534 <int>, f3544 <int>, f4554 <int>, f5564 <int>, f65 <int>,*

*#> # fu <int>*

First we use pivot\_longer() to gather up the non-variable columns:

tb2 <- tb %>%

pivot\_longer(

!c(iso2, year),

names\_to = "demo",

values\_to = "n",

values\_drop\_na = TRUE

)

tb2

*#> # A tibble: 35,750 × 4*

*#> iso2 year demo n*

*#> <chr> <int> <chr> <int>*

*#> 1 AD 1996 m014 0*

*#> 2 AD 1996 m1524 0*

*#> 3 AD 1996 m2534 0*

*#> 4 AD 1996 m3544 4*

*#> 5 AD 1996 m4554 1*

*#> 6 AD 1996 m5564 0*

*#> # … with 35,744 more rows*

Column headers in this format are often separated by a non-alphanumeric character (e.g. ., -, \_, :), or have a fixed width format, like in this dataset. separate() makes it easy to split a compound variables into individual variables. You can either pass it a regular expression to split on (the default is to split on non-alphanumeric columns), or a vector of character positions. In this case we want to split after the first character:

tb3 <- tb2 %>%

separate(demo, c("sex", "age"), 1)

tb3

*#> # A tibble: 35,750 × 5*

*#> iso2 year sex age n*

*#> <chr> <int> <chr> <chr> <int>*

*#> 1 AD 1996 m 014 0*

*#> 2 AD 1996 m 1524 0*

*#> 3 AD 1996 m 2534 0*

*#> 4 AD 1996 m 3544 4*

*#> 5 AD 1996 m 4554 1*

*#> 6 AD 1996 m 5564 0*

*#> # … with 35,744 more rows*

Storing the values in this form resolves a problem in the original data. We want to compare rates, not counts, which means we need to know the population. In the original format, there is no easy way to add a population variable. It has to be stored in a separate table, which makes it hard to correctly match populations to counts. In tidy form, adding variables for population and rate is easy because they’re just additional columns.

In this case, we could also do the transformation in a single step by supplying multiple column names to names\_to and also supplying a grouped regular expression to names\_pattern:

tb %>% pivot\_longer(

!c(iso2, year),

names\_to = c("sex", "age"),

names\_pattern = "(.)(.+)",

values\_to = "n",

values\_drop\_na = TRUE

)

*#> # A tibble: 35,750 × 5*

*#> iso2 year sex age n*

*#> <chr> <int> <chr> <chr> <int>*

*#> 1 AD 1996 m 014 0*

*#> 2 AD 1996 m 1524 0*

*#> 3 AD 1996 m 2534 0*

*#> 4 AD 1996 m 3544 4*

*#> 5 AD 1996 m 4554 1*

*#> 6 AD 1996 m 5564 0*

*#> # … with 35,744 more rows*

**Variables are stored in both rows and columns**

The most complicated form of messy data occurs when variables are stored in both rows and columns. The code below loads daily weather data from the Global Historical Climatology Network for one weather station (MX17004) in Mexico for five months in 2010.

weather <- as\_tibble(read.csv("weather.csv", stringsAsFactors = FALSE))

weather

*#> # A tibble: 22 × 35*

*#> id year month element d1 d2 d3 d4 d5 d6 d7 d8*

*#> <chr> <int> <int> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>*

*#> 1 MX17004 2010 1 tmax NA NA NA NA NA NA NA NA*

*#> 2 MX17004 2010 1 tmin NA NA NA NA NA NA NA NA*

*#> 3 MX17004 2010 2 tmax NA 27.3 24.1 NA NA NA NA NA*

*#> 4 MX17004 2010 2 tmin NA 14.4 14.4 NA NA NA NA NA*

*#> 5 MX17004 2010 3 tmax NA NA NA NA 32.1 NA NA NA*

*#> 6 MX17004 2010 3 tmin NA NA NA NA 14.2 NA NA NA*

*#> # … with 16 more rows, and 23 more variables: d9 <lgl>, d10 <dbl>, d11 <dbl>,*

*#> # d12 <lgl>, d13 <dbl>, d14 <dbl>, d15 <dbl>, d16 <dbl>, d17 <dbl>,*

*#> # d18 <lgl>, d19 <lgl>, d20 <lgl>, d21 <lgl>, d22 <lgl>, d23 <dbl>,*

*#> # d24 <lgl>, d25 <dbl>, d26 <dbl>, d27 <dbl>, d28 <dbl>, d29 <dbl>,*

*#> # d30 <dbl>, d31 <dbl>*

It has variables in individual columns (id, year, month), spread across columns (day, d1-d31) and across rows (tmin, tmax) (minimum and maximum temperature). Months with fewer than 31 days have structural missing values for the last day(s) of the month.

To tidy this dataset we first use pivot\_longer to gather the day columns:

weather2 <- weather %>%

pivot\_longer(

d1:d31,

names\_to = "day",

values\_to = "value",

values\_drop\_na = TRUE

)

weather2

*#> # A tibble: 66 × 6*

*#> id year month element day value*

*#> <chr> <int> <int> <chr> <chr> <dbl>*

*#> 1 MX17004 2010 1 tmax d30 27.8*

*#> 2 MX17004 2010 1 tmin d30 14.5*

*#> 3 MX17004 2010 2 tmax d2 27.3*

*#> 4 MX17004 2010 2 tmax d3 24.1*

*#> 5 MX17004 2010 2 tmax d11 29.7*

*#> 6 MX17004 2010 2 tmax d23 29.9*

*#> # … with 60 more rows*

For presentation, I’ve dropped the missing values, making them implicit rather than explicit. This is ok because we know how many days are in each month and can easily reconstruct the explicit missing values.

We’ll also do a little cleaning:

weather3 <- weather2 %>%

mutate(day = as.integer(gsub("d", "", day))) %>%

select(id, year, month, day, element, value)

weather3

*#> # A tibble: 66 × 6*

*#> id year month day element value*

*#> <chr> <int> <int> <int> <chr> <dbl>*

*#> 1 MX17004 2010 1 30 tmax 27.8*

*#> 2 MX17004 2010 1 30 tmin 14.5*

*#> 3 MX17004 2010 2 2 tmax 27.3*

*#> 4 MX17004 2010 2 3 tmax 24.1*

*#> 5 MX17004 2010 2 11 tmax 29.7*

*#> 6 MX17004 2010 2 23 tmax 29.9*

*#> # … with 60 more rows*

This dataset is mostly tidy, but the element column is not a variable; it stores the names of variables. (Not shown in this example are the other meteorological variables prcp (precipitation) and snow (snowfall)). Fixing this requires widening the data: pivot\_wider() is inverse of pivot\_longer(), pivoting element and value back out across multiple columns:

weather3 %>%

pivot\_wider(names\_from = element, values\_from = value)

*#> # A tibble: 33 × 6*

*#> id year month day tmax tmin*

*#> <chr> <int> <int> <int> <dbl> <dbl>*

*#> 1 MX17004 2010 1 30 27.8 14.5*

*#> 2 MX17004 2010 2 2 27.3 14.4*

*#> 3 MX17004 2010 2 3 24.1 14.4*

*#> 4 MX17004 2010 2 11 29.7 13.4*

*#> 5 MX17004 2010 2 23 29.9 10.7*

*#> 6 MX17004 2010 3 5 32.1 14.2*

*#> # … with 27 more rows*

This form is tidy: there’s one variable in each column, and each row represents one day.

**Multiple types in one table**

Datasets often involve values collected at multiple levels, on different types of observational units. During tidying, each type of observational unit should be stored in its own table. This is closely related to the idea of database normalisation, where each fact is expressed in only one place. It’s important because otherwise inconsistencies can arise.

The billboard dataset actually contains observations on two types of observational units: the song and its rank in each week. This manifests itself through the duplication of facts about the song: artist is repeated many times.

This dataset needs to be broken down into two pieces: a song dataset which stores artist and song name, and a ranking dataset which gives the rank of the song in each week. We first extract a song dataset:

song <- billboard3 %>%

distinct(artist, track) %>%

mutate(song\_id = row\_number())

song

*#> # A tibble: 317 × 3*

*#> artist track song\_id*

*#> <chr> <chr> <int>*

*#> 1 2 Pac Baby Don't Cry (Keep... 1*

*#> 2 2Ge+her The Hardest Part Of ... 2*

*#> 3 3 Doors Down Kryptonite 3*

*#> 4 3 Doors Down Loser 4*

*#> 5 504 Boyz Wobble Wobble 5*

*#> 6 98^0 Give Me Just One Nig... 6*

*#> # … with 311 more rows*

Then use that to make a rank dataset by replacing repeated song facts with a pointer to song details (a unique song id):

rank <- billboard3 %>%

left\_join(song, c("artist", "track")) %>%

select(song\_id, date, week, rank)

rank

*#> # A tibble: 5,307 × 4*

*#> song\_id date week rank*

*#> <int> <date> <int> <dbl>*

*#> 1 1 2000-02-26 1 87*

*#> 2 1 2000-03-04 2 82*

*#> 3 1 2000-03-11 3 72*

*#> 4 1 2000-03-18 4 77*

*#> 5 1 2000-03-25 5 87*

*#> 6 1 2000-04-01 6 94*

*#> # … with 5,301 more rows*

You could also imagine a week dataset which would record background information about the week, maybe the total number of songs sold or similar “demographic” information.

Normalisation is useful for tidying and eliminating inconsistencies. However, there are few data analysis tools that work directly with relational data, so analysis usually also requires denormalisation or the merging the datasets back into one table.

**One type in multiple tables**

It’s also common to find data values about a single type of observational unit spread out over multiple tables or files. These tables and files are often split up by another variable, so that each represents a single year, person, or location. As long as the format for individual records is consistent, this is an easy problem to fix:

1. Read the files into a list of tables.
2. For each table, add a new column that records the original file name (the file name is often the value of an important variable).
3. Combine all tables into a single table.

Purrr makes this straightforward in R. The following code generates a vector of file names in a directory (data/) which match a regular expression (ends in .csv). Next we name each element of the vector with the name of the file. We do this because will preserve the names in the following step, ensuring that each row in the final data frame is labeled with its source. Finally, map\_dfr() loops over each path, reading in the csv file and combining the results into a single data frame.

library(purrr)

paths <- dir("data", pattern = "\\.csv$", full.names = TRUE)

names(paths) <- basename(paths)

map\_dfr(paths, read.csv, stringsAsFactors = FALSE, .id = "filename")

Global Knitr Option of Tidyhydat Package

---

```{r options, include=FALSE}

knitr::opts\_chunk$set(echo = TRUE,

warning = FALSE,

message = FALSE,

eval = nzchar(Sys.getenv("hydat\_eval")),

fig.width=7, fig.height=7)

```

## Package loading

In addition to tidyhydat, this vignette makes use of the [dplyr](https://dplyr.tidyverse.org/) package for data manipulations and [ggplot2](https://ggplot2.tidyverse.org/) for plotting.

```{r packages, warning=FALSE, message=FALSE, echo = TRUE}

library(tidyhydat)

library(dplyr)

library(ggplot2)

```

# `tidyhydat` package

This vignette will outline a few key options that will hopefully make `tidyhydat` useful.

## HYDAT download

To use many of the functions in the `tidyhydat` package you will need to download a version of the HYDAT database, Environment and Climate Change Canada's database of historical hydrometric data then tell R where to find the database. Conveniently `tidyhydat` does all this for you via:

```{r, eval=FALSE}

download\_hydat()

```

This downloads the most recent version of HYDAT and then saves it in a location on your computer where `tidyhydat`'s function will look for it. Do be patient though as this takes a long time! To see where HYDAT was saved you can run `hy\_dir()`. Now that you have HYDAT downloaded and ready to go, you are all set to begin some hydrologic analysis.

## Usage

Most functions in `tidyhydat` follow a common argument structure. We will use the `hy\_daily\_flows()` function for the following examples though the same approach applies to most functions in the package (See `ls("package:tidyhydat")` for a list of exported objects). Much of the functionality of `tidyhydat` originates with the choice of hydrometric stations that you are interested in. A user will often find themselves creating vectors of station numbers. There are several ways to do this.

The simplest case is if you would like to extract only station. You can supply this directly to the `station\_number` argument:

```{r example1, warning=FALSE}

hy\_daily\_flows(station\_number = "08LA001")

```

Another method is to use `hy\_stations()` to generate your vector which is then given the `station\_number` argument. For example, we could take a subset for only those active stations within Prince Edward Island (Province code:PE) and then create vector for `hy\_daily\_flows()`:

```{r example2, warning=FALSE}

PEI\_stns <- hy\_stations() %>%

filter(HYD\_STATUS == "ACTIVE") %>%

filter(PROV\_TERR\_STATE\_LOC == "PE") %>%

pull\_station\_number()

PEI\_stns

hy\_daily\_flows(station\_number = PEI\_stns)

```

We can also merge our station choice and data extraction into one unified pipe which accomplishes a single goal. For example if for some reason we wanted all the stations in Canada that had the name "Canada" in them we unify that selection and data extraction process into a single pipe:

```{r, example3}

search\_stn\_name("canada") %>%

pull\_station\_number() %>%

hy\_daily\_flows()

```

We saw above that if we were only interested in a subset of dates we could use the `start\_date` and `end\_date` arguments. A date must be supplied to both these arguments in the form of YYYY-MM-DD. If you were interested in all daily flow data from station number "08LA001" for 1981, you would specify all days in 1981 :

```{r warning=FALSE, warning=FALSE, message=FALSE, eval=FALSE}

hy\_daily\_flows(station\_number = "08LA001",

start\_date = "1981-01-01", end\_date = "1981-12-31")

```

This generally outlines the usage of the HYDAT functions within `tidyhydat`.

## Real-time functions

In addition to the approved and vetted data in the HYDAT database ECCC also offers unapproved data that is subject to revision. `tidyhydat` provides three functions to access these data sources. Remember these are \*\*unapproved\*\* data and should treated as such:

- `realtime\_stations()`

- `realtime\_dd()`

Not every stations is currently part of the real-time network. Therefore `realtime\_stations()` points to a (hopefully) updated ECCC data file of active real-time stations. We can use the `realtime\_stations()` functionality to get a vector of stations by jurisdiction. For example, we can choose all the stations in Prince Edward Island using the following:

```{r, eval=FALSE}

realtime\_stations(prov\_terr\_state\_loc = "PE")

```

`hy\_stations()` and `realtime\_stations()` perform similar tasks albeit on different data sources. `hy\_stations()` extracts directly from HYDAT. In addition to real-time stations, `hy\_stations()` outputs discontinued and non-real-time stations:

```{r stations, eval=FALSE}

hy\_stations(prov\_terr\_state\_loc = "PE")

```

This is contrast to `realtime\_stations()` which downloads all real-time stations. Though this is not always the case, it is best to use `realtime\_stations()` when dealing with real-time data and `hy\_stations()` when interacting with HYDAT. It is also appropriate to filter the output of `hy\_stations()` by the `REAL\_TIME` column.

### Meterological Service of Canada datamart - `realtime\_dd()`

To download real-time data using the datamart we can use approximately the same conventions discussed above. Using `realtime\_dd()` we can easily select specific stations by supplying a station of interest:

```{r, eval=FALSE}

realtime\_dd(station\_number = "08LG006")

```

Another option is to provide simply the province as an argument and download all stations from that province:

```{r, eval=FALSE}

realtime\_dd(prov\_terr\_state\_loc = "PE")

```

## Search functions

You can also make use of auxiliary functions in `tidyhydat` called `search\_stn\_name()` and `search\_stn\_number()` to look for matches when you know part of a name of a station. For example:

```{r, echo=TRUE}

search\_stn\_name("liard")

```

Similarly, `search\_stn\_number()` can be useful if you are interested in all stations from the \*08MF\* sub-sub-drainage:

```{r, echo=TRUE}

search\_stn\_number("08MF")

```

## Using joins

Sometimes it is required to make use of information from two tables from HYDAT. In some cases, we need to combine the information into one table using a common column. Here we will illustrate calculating runoff by combining the `hy\_stations` tables with the `hy\_daily\_flows` table by the `STATION\_NUMBER` column:

```{r}

stns <- c("08NH130", "08NH005")

runoff\_data <- hy\_daily\_flows(station\_number = stns, start\_date = "2000-01-01") %>%

left\_join(

hy\_stations(station\_number = stns) %>%

select(STATION\_NUMBER, STATION\_NAME, DRAINAGE\_AREA\_GROSS),

by = "STATION\_NUMBER") %>%

## conversion to mm/d

mutate(runoff = Value / DRAINAGE\_AREA\_GROSS \* 86400 / 1e6 \* 1e3)

ggplot(runoff\_data) +

geom\_line(aes(x = Date, y = runoff, colour = STATION\_NAME)) +

labs(y = "Mean daily runoff [mm/d]") +

theme\_minimal() +

theme(legend.position = "bottom")

```

TidyHydat – An Introduction

## Package loading

library(tidyhydat)

library(dplyr)

library(ggplot2)

# tidyhydat package

This vignette will outline a few key options that will hopefully make tidyhydat useful.

## HYDAT download

To use many of the functions in the tidyhydat package you will need to download a version of the HYDAT database, Environment and Climate Change Canada’s database of historical hydrometric data then tell R where to find the database. Conveniently tidyhydat does all this for you via:

download\_hydat()

This downloads the most recent version of HYDAT and then saves it in a location on your computer where tidyhydat’s function will look for it. Do be patient though as this takes a long time! To see where HYDAT was saved you can run hy\_dir(). Now that you have HYDAT downloaded and ready to go, you are all set to begin some hydrologic analysis.

## Usage

Most functions in tidyhydat follow a common argument structure. We will use the hy\_daily\_flows() function for the following examples though the same approach applies to most functions in the package (See ls("package:tidyhydat") for a list of exported objects). Much of the functionality of tidyhydat originates with the choice of hydrometric stations that you are interested in. A user will often find themselves creating vectors of station numbers. There are several ways to do this.

The simplest case is if you would like to extract only station. You can supply this directly to the station\_number argument:

hy\_daily\_flows(station\_number = "08LA001")

## Queried from version of HYDAT released on 2021-07-21

## Observations: 29,890

## Measurement flags: 5,922

## Parameter(s): Flow

## Date range: 1914-01-01 to 2017-12-31

## Station(s) returned: 1

## Stations requested but not returned:

## All stations returned.

## # A tibble: 29,890 x 5

## STATION\_NUMBER Date Parameter Value Symbol

## <chr> <date> <chr> <dbl> <chr>

## 1 08LA001 1914-01-01 Flow 144 <NA>

## 2 08LA001 1914-01-02 Flow 144 <NA>

## 3 08LA001 1914-01-03 Flow 144 <NA>

## 4 08LA001 1914-01-04 Flow 140 <NA>

## 5 08LA001 1914-01-05 Flow 140 <NA>

## 6 08LA001 1914-01-06 Flow 136 <NA>

## 7 08LA001 1914-01-07 Flow 136 <NA>

## 8 08LA001 1914-01-08 Flow 140 <NA>

## 9 08LA001 1914-01-09 Flow 140 <NA>

## 10 08LA001 1914-01-10 Flow 140 <NA>

## # ... with 29,880 more rows

Another method is to use hy\_stations() to generate your vector which is then given the station\_number argument. For example, we could take a subset for only those active stations within Prince Edward Island (Province code:PE) and then create vector for hy\_daily\_flows():

PEI\_stns <- hy\_stations() %>%

filter(HYD\_STATUS == "ACTIVE") %>%

filter(PROV\_TERR\_STATE\_LOC == "PE") %>%

pull\_station\_number()

PEI\_stns

## [1] "01CA003" "01CB002" "01CB004" "01CB018" "01CC002" "01CC005" "01CC010"

## [8] "01CC011" "01CD005"

hy\_daily\_flows(station\_number = PEI\_stns)

## Queried from version of HYDAT released on 2021-07-21

## Observations: 113,141

## Measurement flags: 20,258

## Parameter(s): Flow

## Date range: 1961-08-01 to 2019-12-31

## Station(s) returned: 9

## Stations requested but not returned:

## All stations returned.

## # A tibble: 113,141 x 5

## STATION\_NUMBER Date Parameter Value Symbol

## <chr> <date> <chr> <dbl> <chr>

## 1 01CA003 1961-08-01 Flow NA <NA>

## 2 01CB002 1961-08-01 Flow NA <NA>

## 3 01CA003 1961-08-02 Flow NA <NA>

## 4 01CB002 1961-08-02 Flow NA <NA>

## 5 01CA003 1961-08-03 Flow NA <NA>

## 6 01CB002 1961-08-03 Flow NA <NA>

## 7 01CA003 1961-08-04 Flow NA <NA>

## 8 01CB002 1961-08-04 Flow NA <NA>

## 9 01CA003 1961-08-05 Flow NA <NA>

## 10 01CB002 1961-08-05 Flow NA <NA>

## # ... with 113,131 more rows

We can also merge our station choice and data extraction into one unified pipe which accomplishes a single goal. For example if for some reason we wanted all the stations in Canada that had the name “Canada” in them we unify that selection and data extraction process into a single pipe:

search\_stn\_name("canada") %>%

pull\_station\_number() %>%

hy\_daily\_flows()

## Queried from version of HYDAT released on 2021-07-21

## Observations: 84,594

## Measurement flags: 25,617

## Parameter(s): Flow

## Date range: 1918-08-01 to 2020-12-31

## Station(s) returned: 7

## Stations requested but not returned:

## All stations returned.

## # A tibble: 84,594 x 5

## STATION\_NUMBER Date Parameter Value Symbol

## <chr> <date> <chr> <dbl> <chr>

## 1 01AK001 1918-08-01 Flow NA <NA>

## 2 01AK001 1918-08-02 Flow NA <NA>

## 3 01AK001 1918-08-03 Flow NA <NA>

## 4 01AK001 1918-08-04 Flow NA <NA>

## 5 01AK001 1918-08-05 Flow NA <NA>

## 6 01AK001 1918-08-06 Flow NA <NA>

## 7 01AK001 1918-08-07 Flow 1.78 <NA>

## 8 01AK001 1918-08-08 Flow 1.78 <NA>

## 9 01AK001 1918-08-09 Flow 1.5 <NA>

## 10 01AK001 1918-08-10 Flow 1.78 <NA>

## # ... with 84,584 more rows

We saw above that if we were only interested in a subset of dates we could use the start\_date and end\_date arguments. A date must be supplied to both these arguments in the form of YYYY-MM-DD. If you were interested in all daily flow data from station number “08LA001” for 1981, you would specify all days in 1981 :

hy\_daily\_flows(station\_number = "08LA001",

start\_date = "1981-01-01", end\_date = "1981-12-31")

This generally outlines the usage of the HYDAT functions within tidyhydat.

## Real-time functions

In addition to the approved and vetted data in the HYDAT database ECCC also offers unapproved data that is subject to revision. tidyhydat provides three functions to access these data sources. Remember these are **unapproved** data and should treated as such:

* realtime\_stations()
* realtime\_dd()

Not every stations is currently part of the real-time network. Therefore realtime\_stations() points to a (hopefully) updated ECCC data file of active real-time stations. We can use the realtime\_stations() functionality to get a vector of stations by jurisdiction. For example, we can choose all the stations in Prince Edward Island using the following:

realtime\_stations(prov\_terr\_state\_loc = "PE")

hy\_stations() and realtime\_stations() perform similar tasks albeit on different data sources. hy\_stations() extracts directly from HYDAT. In addition to real-time stations, hy\_stations() outputs discontinued and non-real-time stations:

hy\_stations(prov\_terr\_state\_loc = "PE")

This is contrast to realtime\_stations() which downloads all real-time stations. Though this is not always the case, it is best to use realtime\_stations() when dealing with real-time data and hy\_stations() when interacting with HYDAT. It is also appropriate to filter the output of hy\_stations() by the REAL\_TIME column.

### Meterological Service of Canada datamart - realtime\_dd()

To download real-time data using the datamart we can use approximately the same conventions discussed above. Using realtime\_dd() we can easily select specific stations by supplying a station of interest:

realtime\_dd(station\_number = "08LG006")

Another option is to provide simply the province as an argument and download all stations from that province:

realtime\_dd(prov\_terr\_state\_loc = "PE")

## Search functions

You can also make use of auxiliary functions in tidyhydat called search\_stn\_name() and search\_stn\_number() to look for matches when you know part of a name of a station. For example:

search\_stn\_name("liard")

## # A tibble: 9 x 5

## STATION\_NUMBER STATION\_NAME PROV\_TERR\_STATE\_~ LATITUDE LONGITUDE

## <chr> <chr> <chr> <dbl> <dbl>

## 1 10AA001 LIARD RIVER AT UPPER CROS~ YT 60.1 -129.

## 2 10AA006 LIARD RIVER BELOW SCURVY ~ YT 60.8 -131.

## 3 10BE001 LIARD RIVER AT LOWER CROS~ BC 59.4 -126.

## 4 10ED001 LIARD RIVER AT FORT LIARD NT 60.2 -123.

## 5 10ED002 LIARD RIVER NEAR THE MOUTH NT 61.7 -121.

## 6 10BE005 LIARD RIVER ABOVE BEAVER ~ BC 59.7 -124.

## 7 10BE006 LIARD RIVER ABOVE KECHIKA~ BC 59.7 -127.

## 8 10ED008 LIARD RIVER AT LINDBERG L~ NT 61.1 -123.

## 9 10GC004 MACKENZIE RIVER ABOVE LIA~ NT 61.9 -121.

Similarly, search\_stn\_number() can be useful if you are interested in all stations from the 08MF sub-sub-drainage:

search\_stn\_number("08MF")

## # A tibble: 51 x 5

## STATION\_NUMBER STATION\_NAME PROV\_TERR\_STATE\_~ LATITUDE LONGITUDE

## <chr> <chr> <chr> <dbl> <dbl>

## 1 08MF005 FRASER RIVER AT HOPE BC 49.4 -121.

## 2 08MF035 FRASER RIVER NEAR AGASSIZ BC 49.2 -122.

## 3 08MF038 FRASER RIVER AT CANNOR BC 49.1 -122.

## 4 08MF040 FRASER RIVER ABOVE TEXAS~ BC 50.6 -122.

## 5 08MF062 COQUIHALLA RIVER BELOW N~ BC 49.5 -121.

## 6 08MF065 NAHATLATCH RIVER BELOW T~ BC 50.0 -122.

## 7 08MF068 COQUIHALLA RIVER ABOVE A~ BC 49.4 -121.

## 8 08MF072 FRASER RIVER AT LAIDLAW BC 49.3 -122.

## 9 08MF073 FRASER RIVER AT HARRISON~ BC 49.2 -122.

## 10 08MF001 ANDERSON RIVER NEAR BOST~ BC 49.8 -121.

## # ... with 41 more rows

## Using joins

Sometimes it is required to make use of information from two tables from HYDAT. In some cases, we need to combine the information into one table using a common column. Here we will illustrate calculating runoff by combining the hy\_stations tables with the hy\_daily\_flows table by the STATION\_NUMBER column:

stns <- c("08NH130", "08NH005")

runoff\_data <- hy\_daily\_flows(station\_number = stns, start\_date = "2000-01-01") %>%

left\_join(

hy\_stations(station\_number = stns) %>%

select(STATION\_NUMBER, STATION\_NAME, DRAINAGE\_AREA\_GROSS),

by = "STATION\_NUMBER") %>%

*## conversion to mm/d*

mutate(runoff = Value / DRAINAGE\_AREA\_GROSS \* 86400 / 1e6 \* 1e3)

ggplot(runoff\_data) +

geom\_line(aes(x = Date, y = runoff, colour = STATION\_NAME)) +

labs(y = "Mean daily runoff [mm/d]") +

theme\_minimal() +

theme(legend.position = "bottom")

