```
{“talk_title":
“starts_with(language): Translating column
name contracts from dplyr to dbt”,
 “talk_author”: {
   “author_name”: “Emily Riederer”,
   “author_twtr”: “@emilyriederer”,
   “author_site”: “emilyriederer.com”
  },
 “talk_forum”: {
   “forum_name”: “NYR”,
   “forum_locn”: “New York City, NY”,
   “forum_date”: “2023-07-14”
  }

}
```

Column naming across large datasets and databases has ample room for inconsistency

Order

first_name
name_last

Abbreviations

nm_first
name_last

Meaning

revenue *(Marketing definition)*
revenue (*Finance definition)*

Case

first_name
nameLast

Encoding

web_or_mobile *(Which is 1? 0?)*

Obfuscation

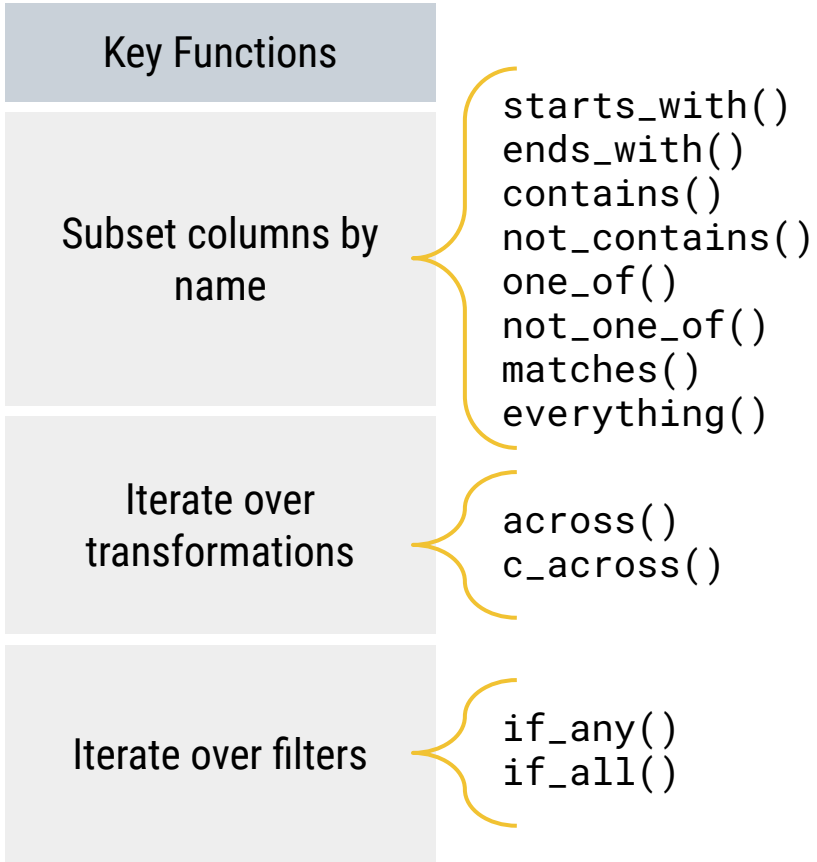
B01001Ce1 *(US Census)*

→ API-inspired column naming

→ columns names as APIs

→ translating to dbt(plyr)

dplyr offers a rich set of “select helpers” to programmatically transform dataset columns



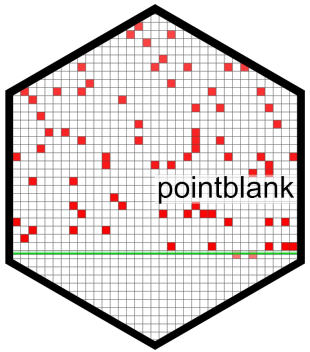
Select helpers incentivize strategic column naming to build more resilient workflows



Defining **data types** for more automated **data transformation**



Encoding **semantics** for more domain-safe **analysis & modeling**



Promising **contracts** on which to conduct **data validation**

Encoding “types” in column names can create DRY-er and more dynamic programming

```
library(dplyr)

marketing_campaign %>%
  group_by(channel) %>%
  summarize(
    across(starts_with("ind"), mean),
    across(contains("spend_pre_"), sum)
  )
```

```
# A tibble: 2 x 5
  channel ind_resubscribe amt_spend_pre_3m amt_spend_pre_6m amt_spend_pre_9m
  <chr>             <dbl>            <dbl>            <dbl>            <dbl>
1 Email              0.25            2026.            2014.            2016.
2 SMS                0.25            2006.            2023.            2004.
```



Encoding “types” in column names can create DRY-er and more dynamic programming

```
library(dplyr)

marketing_campaign %>%
  group_by(channel) %>%
  summarize(
    across(starts_with("ind"), mean, .names = "{gsub('ind', 'prop', {.col})}"),
    across(contains("spend_pre_"), sum)
  )
```

```
# A tibble: 2 x 5
  channel prop_resubscribe amt_spend_pre_3m amt_spend_pre_6m amt_spend_pre_9m
  <chr>              <dbl>            <dbl>            <dbl>            <dbl>
1 Email               0.25            2026.            2014.            2016.
2 SMS                 0.25            2006.            2023.            2004.
```



Encoding semantics can backstop modeling mistakes (e.g. feature leakage)

```
library(recipes)

colnames(marketing_campaign)
```

```
"ind_resubscribe"   "id_customer"
"amt_spend_pre_3m"  "amt_spend_pre_6m"  "amt_spend_pre_9m"
"n_trxn_pre_3m"     "n_trxn_pre_6m"     "n_trxn_pre_9m"
"amt_spend_post_3m" "amt_spend_post_6m" "amt_spend_post_9m"
"n_trxn_post_3m"    "n_trxn_post_6m"    "n_trxn_post_9m"
```

```
rec <-
  recipe(ind_resubscribe ~ ., data = marketing_campaign) %>%
  update_role(starts_with("id_"), new_role = "id variable") %>%
  update_role(contains("_post_"), new_role = "post-target performance")

rec
```

```
Recipe
Inputs:
                      role #variables
             id variable          1
                 outcome          1
 post-target performance          6
               predictor          6
```



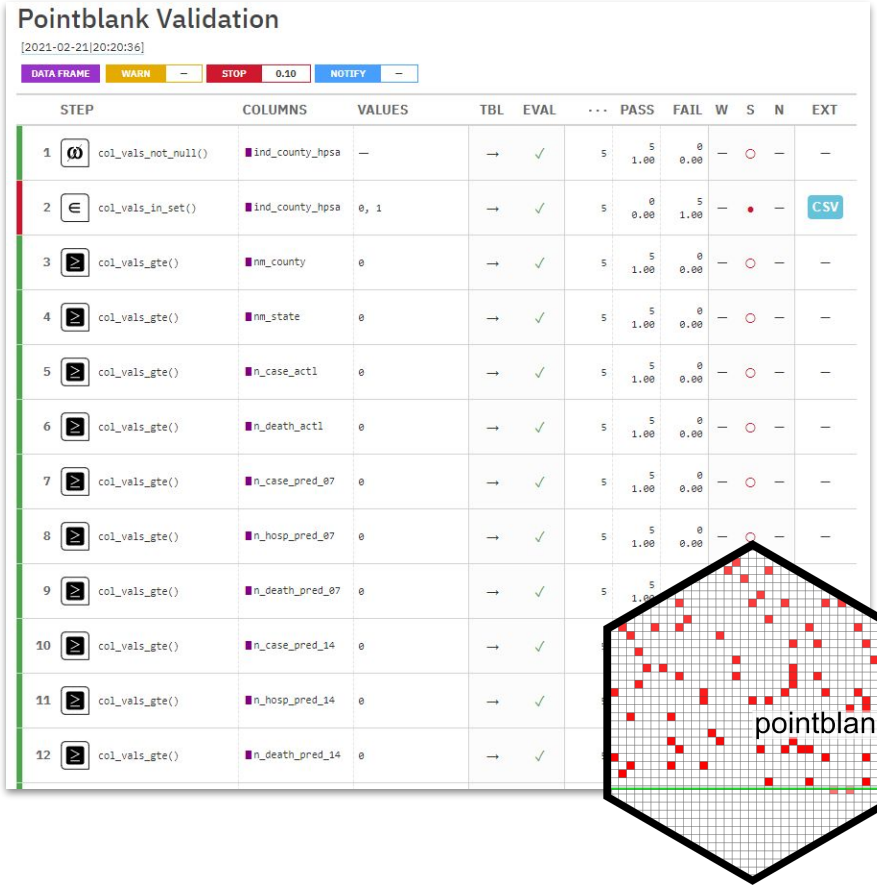
Encoding contracts helps to scale data validation

```
library(pointblank)

create_agent(
  data,
  actions = action_levels(stop_at = 0.1)
  ) %>%

  col_vals_gte(starts_with("N"), 0) %>%
  col_vals_not_null(starts_with("IND")) %>%
  col_vals_in_set(starts_with("IND"), c(0,1)) %>%
  col_is_date(starts_with("DT")) %>%

interrogate()
```



The same paradigm has spread to popular python libraries




```
import pandas as pd

df = pd.DataFrame({“n_a”:[1,2],”n_b”:[3,4], “ind_c”:[0,1]})
cols_n = [c for c in df.columns if c[0:2] == “n_”]
df[cols_n]
```




```
import polars as pl
import polars.selectors as cs

df = pl.DataFrame({“n_a”:[1,2],”n_b”:[3,4], “ind_c”:[0,1]})
df.select( cs.starts_with(“n_”) )
```

→ API-inspired column naming

→ columns names as APIs

→ translating to dbt(plyr)

Column names are the user interface of our data

A	B	C	D
1	10	11	1
2	20	12	10
3	30	13	100
4	40	14	1,000
5	50	15	10,000
...	...	...	...



User Interface



Functionality

Build a data interface with a controlled vocabulary

1. Define simple stubs

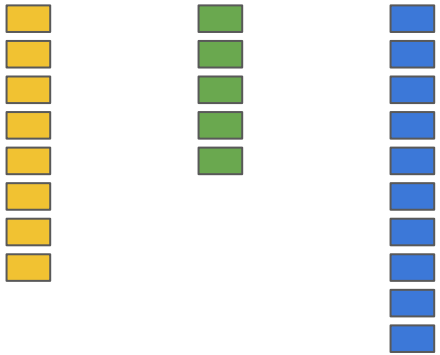
```
stub = semantics + contracts
```



What?
How?

Who?
Where?

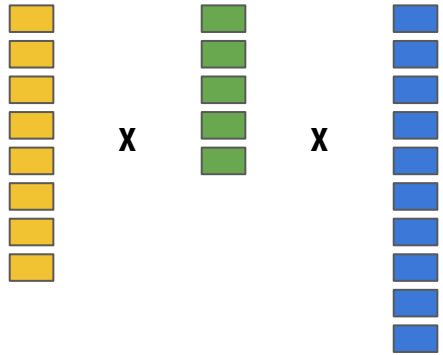
Why?



2. Explain complex concepts

```
name = (type 1 stub)_(type 2 stub)_...
```





An example vocabulary

Stub
ID
IND / IS
BIN
N
DT
...

An example vocabulary

Stub	Semantics
`ID`	Unique entity identifier
`IND / IS`	Binary 0/1 indicator; rest of name describes 1 condition
`BIN`	Binary 0/1 indicator; rest of name describes 1 condition
`N`	Count of quantity or event occurrences
`DT`	Date of an event
`...`	...

An example vocabulary

Stub	Semantics
`ID`	Unique entity identifier
`IND / IS`	Binary 0/1 indicator; rest of name describes 1 condition
`BIN`	Binary 0/1 indicator; rest of name describes 1 condition
`N`	Count of quantity or event occurrences
`DT`	Date of an event
`...`	...

An example vocabulary

Stub	Semantics	Contracts
`ID`	Unique entity identifier	Numeric, primary / surrogate key
`IND / IS`	Binary 0/1 indicator; rest of name describes 1 condition	Always 0 or 1, non-null
`BIN`	Binary 0/1 indicator; rest of name describes 1 condition	Always 0 or 1
`N`	Count of quantity or event occurrences	Non-negative integer, non-null
`DT`	Date of an event	Date, ISO 8601 (YYYY-MM-DD)
`...`	...	...

An example vocabulary

Stub	Semantics	Contracts
`ID`	Unique entity identifier	Numeric, primary / surrogate key
`IND / IS`	Binary 0/1 indicator; rest of name describes 1 condition	Always 0 or 1, **non-null**
`BIN`	Binary 0/1 indicator; rest of name describes 1 condition	Always 0 or 1
`N`	Count of quantity or event occurrences	Non-negative integer, non-null
`DT`	Date of an event	Date, **ISO 8601** (YYYY-MM-DD)
`...`	...	...

An example vocabulary

Stub
USER
LOGIN
...

An example vocabulary

Stub
USER
LOGIN?
...

An example vocabulary

Stub	Semantics
`USER`	Unique site visitor as determined by IP address
`LOGIN`	A successful authentication (password, MFA) by a confirmed human actor (**after passing Captcha**)
`...`	…

An example vocabulary

Stub	Semantics	Consequence
`USER`	Unique site visitor as determined by IP address	Does **not uniquely identify a person across devices**
`LOGIN`	A successful authentication (password, MFA) by a confirmed human actor (**after passing Captcha**)	
`...`	…	…

An example vocabulary

Stub	Semantics	Consequence
`USER`	Unique site visitor as determined by IP address	Does **not uniquely identify a person across devices**
`LOGIN`	A successful authentication (password, MFA) by a confirmed human actor (**after passing Captcha**) ~~A session beginning with a visit to the login screen~~ ~~The click of the login button after typing username and password~~	
`...`	…	…

An example vocabulary

Types
ID
IND / IS
BIN
N
AMT
VAL
DT
TM
CAT
...

X

Subjects
USER
LOGIN
SESSION
CLICK
...

X

Details
UTM
DURATION
...

```
{DT | TM}_{LOGIN | SESSION}

ID_{USER | SESSION | LOGIN | VIEW}

{CAT | CD}_SOURCE_UTM

{CAT | CD}_MEDIUM_UTM

AMT_{SESSION | VIEW}_DURATION

...
```

Encoding semantics can backstop analytical accuracy

Passed Captcha?	LOGIN ✗	IND_LOGIN ✓	BIN_LOGIN ✓
No	*NA*	0	*NA*
No	*NA*	0	*NA*
Yes	0	0	0
Yes	1	1	1
Yes	1	1	1

P(Login) P(Login|Captcha)

```
summarize(
  logins,
  across(everything(), mean, na.rm = TRUE)
)
```

```
      login ind_login bin_login
1 0.6666667       0.4 0.6666667
```

```
summarize(
  logins,
  across(starts_with('ind_'), mean, na.rm = TRUE)
)
```

```
  ind_login
1       0.4
```



→ API-inspired column naming

→ columns names as APIs

→ translating to dbt(plyr)

dbt brings software engineering best practices to SQL-based workflows

Clean Code

- Jinja-templated SQL offers variables, control flow
- Macros encapsulate reused logic

Organized Projects

- Monolithic pipelines decomposed into atomic queries
- Ability to test and version control

Developer Workflow

- Isolated dev / prod environments
- In-pipeline testing
- Automated orchestration
- Logging



dbt templating affords SQL with richer programming logic like variables and control flow

```
select

  coalesce(a, 0) as a,
  coalesce(b, 0) as b,
  coalesce(c, 0) as c,

  (a - lag(a,1) as w) / lag(a,1) over w as a_yoy,
  (b - lag(b,1) as w) / lag(b,1) over w as b_yoy,
  (c - lag(c,1) as w) / lag(c,1) over w as c_yoy

from my_db.my_schema.my_table
```

```
{% set vars = ['a','b','c']  %}

select

  {% for v in vars %}

  coalesce({{v}}, 0) as {{v}},

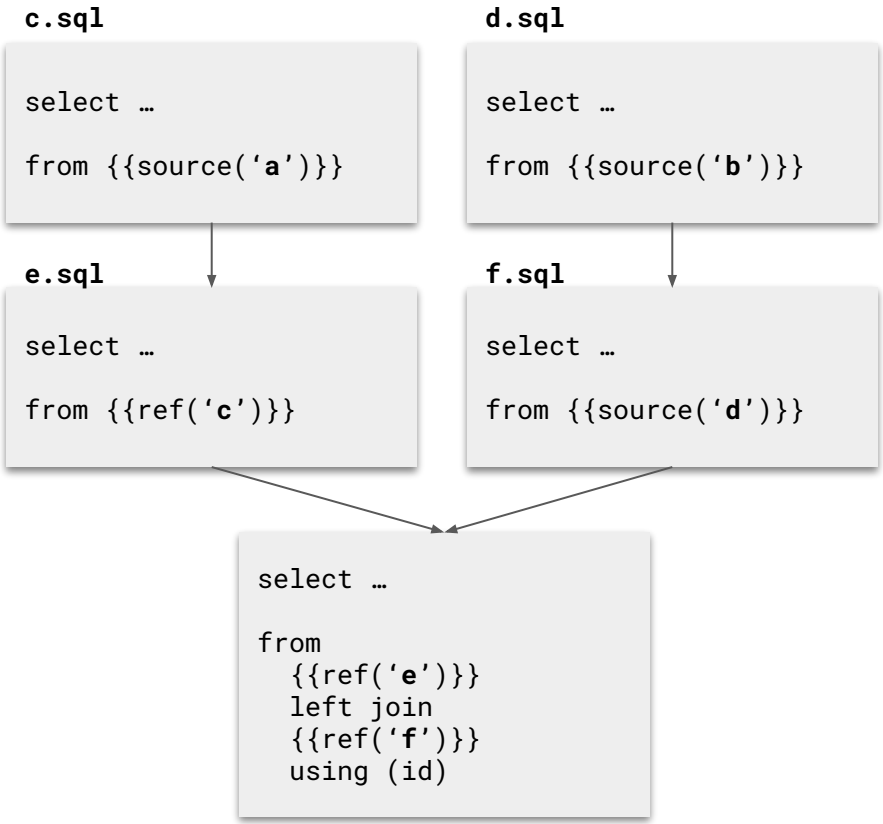
  yoy( {{v}} ) as {{v}}_yoy

  {% endfor %}

from {{ ref('my_table') }}
```

1. **Variables** for iteration
2. **Control flow** with Jinja templating
3. **Macros** to encapsulate logic for DRY scripts
4. **Dynamic tables refs** to support testing & orchestration

dbt modularizes SQL data pipelines into testable and version controllable components



What You Do

- Write atomic SELECT statements
- Isolate reusable logic in functions
- Insert tests throughout pipeline

What You Don't

- Orchestration - dbt infers dependency graph
- Manage pipelines in monolithic notebooks
- Write 'glue' code (e.g. updates, inserts, grants)

dbtplyr brings dplyr's select helpers to dbt to support column-name contracts

Key Functions	

Subset columns by name	`starts_with()`
	`ends_with()`
	`contains()`
	`not_contains()`
	`one_of()`
	`not_one_of()`
	`matches()`
	`everything()`

Iterate over transformations	`across()` `c_across()`
Iterate over filters	`if_any()` `if_all()`

inspired by R's dplyr syntax!



dbtplyr brings dplyr's select helpers to dbt to support column-name contracts

Key Functions
Subset columns by name
Iterate over transformations
Iterate over filters

```
{% set cols =
        dbtplyr.get_column_names(ref('data')) %}
{% set cols_ind =
        dbtplyr.starts_with(cols, 'ind') %}
{% set cols_notnull = ['x', 'y'] %}
```

```
['x','y','ind_a','ind_b']
```

dbtplyr brings dplyr's select helpers to dbt to support column-name contracts

Key Functions

Subset columns by name

```
{% set cols =
        dbtplyr.get_column_names(ref('data')) %}
{% set cols_ind =
        dbtplyr.starts_with(cols, 'ind') %}
{% set cols_notnull = ['x', 'y'] %}
```

```
['x','y','ind_a','ind_b']
```

Iterate over transformations

```
select
  {{ dbtplyr.across(
        cols_ind,
        "avg({var}) as prop_{var}") }}
from {{ ref('data') }}
```

```
select
 avg(ind_a) as prop_ind_a,
 avg(ind_b) as prop_ind_b
from {{ ref('data') }}
```

Iterate over filters

dbtplyr brings dplyr's select helpers to dbt to support column-name contracts

Key Functions

Subset columns by name

```
{% set cols =
        dbtplyr.get_column_names(ref('data')) %}
{% set cols_ind =
        dbtplyr.starts_with(cols, 'ind') %}
{% set cols_notnull = ['x', 'y'] %}
```

```
['x','y','ind_a','ind_b']
```

Iterate over transformations

```
select
  {{ dbtplyr.across(
        cols_ind,
        "avg({var}) as prop_{var}") }}
from {{ ref('data') }}
```

```
select
 avg(ind_a) as prop_ind_a,
 avg(ind_b) as prop_ind_b
from {{ ref('data') }}
```

Iterate over filters

```
where

 {{ dbtplyr.if_all(
       cols_notnull,
       "not {var} is null") }}
```

```
where

  not x is null and
  not y is null
```

Broken contracts frustrate users

ID_VARIANT	N_CLICK_07	N_CLICK_14	N_CLIK_21	N_28_CLICK
1	100	172	202	291
2	112	136	154	191
3	156	181	202	235

```
select
  n_click_07,
  n_clik_14..?
from table
```

Jinja templates enforce consistent naming and definitions

```
select

  id_variant,

    count_if(n_days <= 07)
        as n_click_07,

    count_if(n_days <= 14)
        as n_click_14
```

Jinja templates enforce consistent naming and definitions

```
{% set lags = [‘07’,’14’,’21’,’24’] %}

select

  id_variant,

  {% for l in var('lags') %}

      count_if(n_days <= {{l}})
          as n_click_{{l}}

  {% if not loop.last %},{% endif %}
  {% endfor %}
```



```
select

  id_variant,

    count_if(n_days <= 07)
        as n_click_07,

    count_if(n_days <= 14)
        as n_click_14
```

Broken contracts lie to users

```
select count(*)

from logins

where dt_login = ‘2021-01-01’
```

DT_LOGIN	ID_LOGIN	IND_LOGIN
2021-01-01T 10:25:28	123	1
2021-01-01T 02:10:53	456	1
2021-01-02T 07:20:00	789	0

DT_LOGIN	ID_LOGIN	IND_LOGIN
2021-01-01	123	1
2021-01-01	456	1
2021-01-02	789	0

Custom macros + dbtplyr enforce contracts systemically

```
select

  cast(n_a as int64) as n_a,
  cast(n_c as int64) as n_c,

  date(dt_b) as dt_b,
  date(dt_d) as dt_d,

  coalesce(ind_b,0) as ind_b,
  coalesce(ind_c,0) as ind_c
```

Custom macros + dbtplyr enforce contracts systemically

```
{% set cols =
        dbtplyr.get_column_names( ref('data') )
%}
{% set cols_n =
        dbtplyr.starts_with(cols, 'n') %}
{% set cols_dt =
        dbtplyr.starts_with(cols, 'dt') %}
{% set cols_ind =
        dbtplyr.starts_with(cols, 'ind') %}

select

  {{ dbtplyr.across(cols_n,
                    "cast({var} as int)
                     as n_{var}")}},
  {{ dbtplyr.across(cols_dt,
                    "date({var})
                     as dt_{var})")}},
  {{ dbtplyr.across(cols_ind,
                    "coalesce({c}, 0)
                     as ind_{var}") }}
```



```
select

  cast(n_a as int64) as n_a,
  cast(n_c as int64) as n_c,

  date(dt_b) as dt_b,
  date(dt_d) as dt_d,

  coalesce(ind_b,0) as ind_b,
  coalesce(ind_c,0) as ind_c
```

Custom macros + dbtplyr enforce contracts systemically

```
{% set cols =
        dbtplyr.get_column_names( ref('data') )
%}
{% set cols_n =
        dbtplyr.starts_with(cols, 'n') %}
{% set cols_dt =
        dbtplyr.starts_with(cols, 'dt') %}
{% set cols_ind =
        dbtplyr.starts_with(cols, 'ind') %}

select

  {{ dbtplyr.across(cols_n,
                    "cast({var} as int)
                     as n_{var}")}},
  {{ dbtplyr.across(cols_dt,
                    "date({var})
                     as dt_{var})")}},
  {{ dbtplyr.across(cols_ind,
                    "coalesce({c}, 0)
                     as ind_{var}") }}
```



```
select

  cast(n_a as int64) as n_a,
  cast(n_c as int64) as n_c,

  date(dt_b) as dt_b,
  date(dt_d) as dt_d,

  coalesce(ind_b,0) as ind_b,
  coalesce(ind_c,0) as ind_c
```

Broken contracts evade detection

```
{{ dbtplyr.across(cols_n, “cast({var} as int) as n_{var}”)}}
```

N_A	N_B
12.00	3.25
19.00	4.67
27.00	8.99



N_A	N_B
12	3
19	5
27	9

Testing confirms any non-enforceable contracts are upheld

```
{% set cols = get_column_names(ref('prep')) %}
{% set cols_n =
        dbtplyr.starts_with(cols, 'n') %}

select *
from {{ ref('my_source') }}
where

 {%- for c in cols_n %}

  abs({{c}} - cast({{c}} as int64)) > 0.01 or

 {% endfor %}

  FALSE
```



```
with dbt__CTE__INTERNAL_test as (

select *
from `db`.`dbt_emily`.`my_source`
where

    abs(n_a - cast(n_a as int64)) > 0.01 or
    abs(n_b - cast(n_b as int64)) > 0.01 or
    abs(n_c - cast(n_c as int64)) > 0.01 or

   FALSE
)

select count(*) from dbt__CTE__INTERNAL_test
```

→ API-inspired column naming

→ columns names as APIs

→ translating to dbt(plyr)

Questions?

↓ Get in touch ↓

@emilyriederer on Web | Twitter | GitHub | LinkedIn | Gmail

↓ Check out these resources ↓

dbt Learning Resources
dbtplyr repo
Blog post with example pipeline
Blog post on column name contracts
Jenny Bryan's talk on Naming Things

```
{“talk_title":
“starts_with(language): Translating column
name contracts from dplyr to dbt”,
 “talk_author”: {
   “author_name”: “Emily Riederer”,
   “author_twtr”: “@emilyriederer”,
   “author_site”: “emilyriederer.com”
  },
 “talk_forum”: {
   “forum_name”: “NYR”,
   “forum_locn”: “New York City, NY”,
   “forum_date”: “2023-07-14”
  }

}
```