PRQL Tutorial: A Modern Query Language

PRQL (Pipeline Query Language) is a modern alternative to SQL.
Benefits: readability, composability, and pipeline-style data transformations, can be used in unix terminal with fish shell.

1. Basic Pipeline Structure

PRQL queries flow from top to bottom like reading English, starting with your data source:

```
from employees
filter department == "Engineering"
select {name, salary}
```

This is equivalent to:

```
SELECT name, salary
FROM employees
WHERE department = 'Engineering'
```

2. Pipeline Operators

PRQL uses the pipe symbol `|` or newlines to chain operations:

```
# Using pipe symbols (one-liner)
from sales | filter amount > 1000 | select {date, amount, customer}

# Using newlines (recommended for complex queries)
from sales
filter amount > 1000
select {date, amount, customer}
```

3. Core Transform Functions

Select - Choose and transform columns

```
from products
select {
  name,
  price,
  discounted_price = price * 0.9,
  category = category | upper
}
```

Filter - Keep rows that match conditions

```
from orders
filter status == "completed"
filter order_date >= @2024-01-01    # Date literal syntax
filter amount > 100
```

Derive - Add new columns

```
from employees
derive {
  full_name = f"{first_name} {last_name}",  # String interpolation
  annual_salary = monthly_salary * 12,
  seniority = case [
    years_experience >= 10 => "Senior",
    years_experience >= 5 => "Mid-level",
    true => "Junior"
  ]
}
```

4. Aggregation and Grouping

Basic aggregation

```
from sales
aggregate {
  total_revenue = sum amount,
  order_count = count this,
  avg_order = average amount
}
```

Group by with aggregation

```
from sales
group customer_id (
  aggregate {
    total_spent = sum amount,
    order_count = count this,
    first_order = min order_date,
    last_order = max order_date
  }
)
```

Multiple grouping levels

```
from sales
group {region, product_category} (
  aggregate {
    revenue = sum amount,
    units_sold = sum quantity
  }
)
```

5. Sorting and Limiting

```
from employees
sort {-salary, +hire_date}  # - for descending, + for ascending
take 10                     # Limit to 10 rows
from products
sort price
take 5..10                  # Skip first 5, take next 5 (offset + limit)
```

6. Joins

```
from employees
join departments (==department_id)
select {
  employees.name,
  employees.salary,
  departments.department_name
}
# Left join with explicit syntax
from orders
join side:left customers (==customer_id)
select {
  orders.order_id,
  orders.amount,
  customers.name ?? "Unknown Customer"  # Null coalescing
}
```

7. Window Functions

```
from sales
group customer_id (
  sort order_date
  window expanding:true (
    derive running_total = sum amount
  )
)
from employees
derive {
  salary_rank = rank salary,
  dept_avg_salary = average salary | group department_id
}
```

8. Functions and Reusability

Define custom functions

```
let top_n = func n column -> (
  sort {-column}
  take n
)

# Use the function
from products
top_n 5 price
```

Function with multiple parameters

```
let sales_summary = func date_col amount_col -> (
  group (date_col | date.month) (
    aggregate {
      monthly_total = sum amount_col,
      order_count = count this
    }
  )
```

```
)

from orders
sales_summary order_date amount
```

9. Advanced Features

Loop (Recursive CTEs)

```
# Generate a date spine
let date_range = (
  from_text format:csv """
  date
  2024-01-01
  """
  loop (
    filter date <= @2024-01-10
    derive date = date + 1
  )
)
```

Working with JSON (using s-strings for SQL dialect features)

```
let json_extract = func json_col key -> s"JSON_EXTRACT({json_col}, '$.{key}')"

from user_events
derive {
  event_type = json_extract event_data "type",
  user_id = json_extract event_data "user_id"
}
```

Null handling

```
from customers
derive {
  display_name = name ?? "Anonymous",           # Null coalescing
  clean_email = email | lower | text.trim,     # Chain transformations
  age_group = case [
    age == null => "Unknown",
    age < 18 => "Minor",
    age >= 65 => "Senior",
    true => "Adult"
  ]
}
```

10. Testing and Literals

PRQL makes testing easy with inline data:

```
# Test your transformations with sample data
from [
  {name: "Alice", age: 30, dept: "Engineering"},
  {name: "Bob", age: 25, dept: "Sales"},
  {name: "Carol", age: 35, dept: "Engineering"}
]
```

```
filter dept == "Engineering"
derive seniority = case [age >= 30 => "Senior", true => "Junior"]
```

11. Practical Example: Sales Analysis

Here's a comprehensive example analyzing sales data:

```
let sales_analysis = (
  from sales
  join customers (==customer_id)
  join products (==product_id)

  # Clean and enrich data
  filter order_date >= @2024-01-01
  derive {
    revenue = quantity * price,
    discount_amount = revenue * discount_rate,
    net_revenue = revenue - discount_amount,
    order_month = order_date | date.month
  }

  # Group by customer and month
  group {customer_id, customers.name, order_month} (
    aggregate {
      total_orders = count this,
      total_revenue = sum net_revenue,
      avg_order_value = average net_revenue,
      total_items = sum quantity
    }
  )

  # Add rankings and percentiles
  derive {
    revenue_rank = rank total_revenue,
    revenue_percentile = percent_rank total_revenue
  }

  # Final formatting
  sort {-total_revenue}
  select {
    customer = name,
    month = order_month,
    orders = total_orders,
    revenue = total_revenue | math.round 2,
    avg_order = avg_order_value | math.round 2,
    rank = revenue_rank
  }
)

sales_analysis
```

12. Key Advantages Recap

1. **Readable**: Top-to-bottom flow like natural language
2. **Composable**: Define functions once, use everywhere
3. **Regular**: Consistent syntax across operations
4. **Modern**: String interpolation, null coalescing, date literals

5. **Testable**: Easy to test with inline data
6. **Portable**: Compiles to SQL for any database

Getting Started

Try PRQL in your browser at the online playground, or install it locally:

```
# Python
pip install pyprql

# CLI tool
cargo install prql-compiler

# VS Code extension
# Search for "PRQL" in the extensions marketplace
```

PRQL maintains all of SQL's power while providing a more ergonomic and maintainable syntax for modern data work.

How to use from Python

Write query in PRQL, compile it into SQL - and run SQL query normally

```
import pyprql
prql = """
  from employees
  filter department == "Engineering"
  derive annual_salary = monthly_salary * 12
  sort {-annual_salary}
  select {name, department, annual_salary}
  take 10
"""

sql = pyprql.compile(prql)

# then use sql to query DB
```

How to use from fish shell

```
cargo install prqlc

# Install fd if needed
brew install fd-find  # macOS
```

Key Fish Functions Created

```
prql2sql - Compile PRQL files to SQL
prqlrun - Compile and execute against SQLite
csvql - Query CSV files directly with PRQL
explore_data - Quick data exploration
validate_prql_files - Check all PRQL files in project
```

Examples:

Example 1

```
csvql sales.csv "from data | group region (aggregate {total = sum amount})"
```

Example 2

```
fd -e csv ./data/ | while read file
    csvql $file "from data | take 5"
end
```

Example 3

```
fd -e prql -x sh -c 'echo -n "Checking {}: "; if prqlc compile {} >/dev/null 2>&1;
then echo "✓ Valid"; else echo "✗ Invalid"; fi'
```

Fish Abbreviations

```
pc  → prqlc compile
pcs → prqlc compile --target sqlite
pcp → prqlc compile --target postgres
```

Usage Tips

- Save functions in ~/.config/fish/functions/ for persistence
- Use fd patterns like fd -e prql to find PRQL files
- Pipe operations work great: echo "query" | prqlc compile
- Combine with other tools like sqlite3, jq, csvkit