

Welcome to Data C88C!

Lecture 23: Designing Functions

Monday, August 4th, 2025

Week 7

Summer 2025

Instructor: Eric Kim (ekim555@berkeley.edu)

Announcements

- Ants project checkpoint due tonight (Mon Aug 4th)
- End-of-semester surveys [[link](#)]
 - If 65% or more students complete both surveys by Friday August 15th at 11:59 PM, then everyone will receive 0.5 point of extra credit! If this goal is not met, nobody will receive the extra point.
 - If 75% or more students complete both surveys by Friday August 15th at 11:59 PM, then everyone will receive 0.5 additional point of extra credit.
- Please submit your "Ask Me Anything" (AMA) questions to Ed: [[link](#)]
- No lab this week!

Final Exam Logistics

- Please read this Ed post very carefully: [\[link\]](#)
 - Final exam: Tuesday August 12th, 3:00pm-5:00pm PT
 - Alternate times
 - Tuesday, August 12th, 7:00 pm-9:00pm PT
 - Wednesday, August 13th, 8:20am-10:20am PT
 - Gradescope online exam, Zoom proctoring
 - Two page handwritten cheatsheet allowed (see Ed post for details)
 - Final exam reference sheet: [\[link\]](#)
 - (same setup as the Midterm)
 - If you can't make any of these times, please fill out the form linked in the above Ed post and let us know **ASAP**
 -
-

Lecture Overview

- SQL
 - Query execution order
 - Subqueries
- (For fun) Python type annotations, type systems

SQL order of execution

SELECT S	3
FROM R1, R2, ...	1
WHERE C1;	2

Order of execution:

1. **FROM**: Fetch the tables and compute the cross product of R1, R2, ...
 2. **WHERE**: "Row filter." For each tuple from step 1, keep only those that satisfy condition C1
 3. **SELECT**: add to output based on S
-

SQL order of execution

SELECT S

FROM R1, R2, ...

WHERE C1;

3

1

2

SELECT flavor, price * 2

FROM cones

WHERE price > 4;

Flavor	Color	Price
strawberry	pink	3.55
chocolate	light brown	4.75
chocolate	dark brown	5.25
strawberry	pink	5.25
bubblegum	pink	4.75

Simulation of SQL query execution (in order)

FROM cones

Flavor	Color	Price
strawberry	pink	3.55
chocolate	light brown	4.75
chocolate	dark brown	5.25
strawberry	pink	5.25
bubblegum	pink	4.75

WHERE price > 4

Flavor	Color	Price
chocolate	light brown	4.75
chocolate	dark brown	5.25
strawberry	pink	5.25
bubblegum	pink	4.75

SELECT flavor, price * 2

Flavor	price * 2
chocolate	9.5
chocolate	10.5
strawberry	10.5
bubblegum	9.5

SQL order of execution

SELECT S	5
FROM R1, R2, ...	1
WHERE C1	2
GROUP BY A1, A2, ...	3
HAVING C2;	4

Tip: aggregations (**GROUP BY**) happen after filtering (**WHERE**)

Order of execution:

1. **FROM**: Fetch the tables and compute the cross product of R1, R2, ...
 2. **WHERE**: "Row filter." For each tuple from step 1, keep only those that satisfy condition C1
 3. **GROUP BY**: For each group, compute all aggregates needed in C2 and S
 4. **HAVING**: For each group, check if C2 is satisfied
 5. **SELECT**: add to output based on S
-

SQL order of execution

SELECT S

FROM R1, R2, ...

WHERE C1

GROUP BY A1, A2, ...

HAVING C2;

5

1

2

3

4

```
SELECT flavor, MAX(price * 2)
FROM cones
WHERE price > 4
GROUP BY flavor
HAVING max(price) - min(price) <= 0.25;
```

Flavor	Color	Price
strawberry	pink	3.55
chocolate	light brown	4.75
chocolate	dark brown	5.25
strawberry	pink	5.25
bubblegum	pink	4.75

Simulation of SQL query execution (in order)

FROM cones

Flavor	Color	Price
strawberry	pink	3.55
chocolate	light brown	4.75
chocolate	dark brown	5.25
strawberry	pink	5.25
bubblegum	pink	4.75



WHERE price > 4

Flavor	Color	Price
chocolate	light brown	4.75
chocolate	dark brown	5.25
strawberry	pink	5.25
bubblegum	pink	4.75

GROUP BY flavor

Flavor	Color	Price
chocolate	light brown	4.75
chocolate	dark brown	5.25
strawberry	pink	5.25
bubblegum	pink	4.75

HAVING max(price) - min(price) <= 0.25

Flavor	Color	Price
strawberry	pink	5.25
bubblegum	pink	4.75

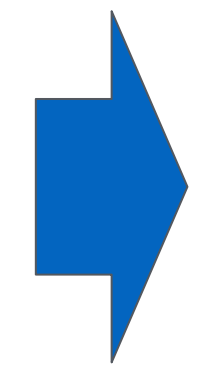
SELECT flavor, MAX(price * 2)

Flavor	max(price) * 2
strawberry	10.5
bubblegum	9.5

SQL execution order

SQL "written" order

SELECT S	5
FROM R1, R2, ...	1
WHERE C1	2
JOIN J1 ON J2	1
GROUP BY A1, A2, ...	3
HAVING C2	4
ORDER BY O1	6
LIMIT L1;	7



SQL execution order

FROM R1, R2, ...	1
JOIN J1 ON J2	1
WHERE C1	2
GROUP BY A1, A2, ...	3
HAVING C2	4
SELECT S	5
ORDER BY O1	6
LIMIT L1;	7

```
SELECT flavor, prices * 2 AS prices_2x
FROM cones
WHERE prices_2x <= 5;
```

```
FROM cones
WHERE prices_2x <= 5
SELECT flavor, prices * 2 AS prices_2x
```

Most SQL implementations will throw an error like "column name `prices_2x` not defined", but this works in sqlite3!
In this course, we will accept this query.


Fun fact: most SQL implementations don't let you use column aliases defined in SELECT in the WHERE clause. But, sqlite3 lets you: [\[link\]](#)

Subqueries ("nested" queries)

```
SELECT flavor, max(price * 2) as max_price_2x
FROM cones
WHERE price > 4
GROUP BY flavor
HAVING max(price) - min(price) <= 0.25;
```

Flavor	max_price_2x
strawberry	10.5
bubblegum	9.5

What if I wanted to do additional processing to the query output? Ex: add an additional filter like: "only fetch rows where `max_price_2x` >= 10"?

 **WHERE max_price_2x >= 10;**

Error: near "WHERE": syntax error

Takeaway: Subqueries have additional forms beyond this one.
In this class (C88C SU25), we'll study only this form of subquery ("FROM" type).

If you're curious to learn more about the other subquery forms, see: [\[link\]](#)

Flavor	Color	Price
strawberry	pink	3.55
chocolate	light brown	4.75
chocolate	dark brown	5.25
strawberry	pink	5.25
bubblegum	pink	4.75

Solution: use a **subquery**:

```
SELECT a.flavor, a.max_price_2x
FROM
(
    SELECT flavor, MAX(price * 2) as max_price_2x
    FROM cones
    WHERE price > 4
    GROUP BY flavor
    HAVING max(price) - min(price) <= 0.25
) a
WHERE
a.max_price_2x >= 10;
```

subquery output table
alias `a`

} Can add more here, like an additional GROUP BY, HAVING, etc...

(reference) SQL in C88C (v2)

- Here is all of the SQL that we cover in C88C that you (the student) are responsible for
 - There is more to SQL than this, but this is a good starting point

```
SELECT select_list  
  
[ FROM table_source(s) ] [ WHERE search_condition ]  
  
[ JOIN table ON join_condition ]  
  
[ GROUP BY group_by_expression ]  
  
[ HAVING search_condition ]  
  
[ ORDER BY order_expression [ ASC | DESC ] ]  
  
[ LIMIT [limit] ];
```

```
# aggregator functions (used with GROUP BY)  
min(), max(), avg(), sum(), count(),  
count(distinct x), count(*)
```

```
# aliasing  
select colA AS colA_alias ...
```

```
CREATE TABLE [table_name] AS  
    SELECT [val1] AS [col1], [val2] AS [col2], ... UNION  
    SELECT [val3], [val4], ... UNION  
    SELECT [val5], [val6], ...;
```

```
# subqueries ("FROM" type only)  
SELECT select_list  
FROM (  
    SELECT ...  
) subquery_alias  
...the rest of your query...
```

```
# operators  
comparison: =, >, <, <=, >=, != (or <>)  
boolean: AND, OR  
arithmetic: +, -, *, /  
concatenation: ||
```

(For fun) Python type annotations ("type hints")

- Python 3 has support for type annotations [\[link\]](#)
- Main idea: syntax to describe the input and output types of your functions and variables

```
def square_nums(nums):  
    if not nums:  
        return []  
    return [nums ** 2] + square_nums(nums[1:])
```



```
def square_nums_anno(nums: list[int]) -> list[int]:  
    if not nums:  
        return []  
    return [nums[0] ** 2] + square_nums_anno(nums[1:])
```

`list[int]` means: a `list` where
each element is type `int`


Parameter `nums` is
of type `list[int]`

Returns a
`list[int]`


(For fun) Python type annotations

- Useful type annotation patterns

`from typing import Optional`

dict with key type `str`,
value type `float`


```
def some_fn(b: float, d: dict[str, float]) -> Optional[float]:  
    if b <= 3.14:  
        return None  
    return d["pi"] * 2.5
```


Returns either a
`float`, or `None`


`from typing import Callable`


A two-arg fn (int, int)
that returns an int


```
def my_hof(fn: Callable[[int, int], int]) -> int:  
    return fn(1, 2)
```

`from typing import Union`

Has type either
`int` or `str`


```
def another_fn(a_int_or_str: Union[int, str]) -> tuple[int, str, float]:  
    a_int = int(a_int_or_str)  
    return a_int, 's', 3.14
```

Returns an `int`, a
`str`, and a `float` (all
in a single tuple)


(For fun) Python type annotations

- Most Python code editors support type checking (including VSCode!), and will warn you if it detects a type violation
- Great way to catch type-related bugs at "compile time" rather than at runtime

```
16
17 def square_nums_anno(nums: list[int]) -> list[int]:
18     if not nums:
19         return None
20     return [nums[0] ** 2] + square_nums_anno(nums[1:])
21
```

Type "None" is not assignable to return type "list[int]"
"None" is not assignable to "list[int]" Pylance([reportReturnType](#))

View Problem (Alt+F8) Quick Fix... (Ctrl+.)

```
19         return None
20     return [nums[0] ** 2] + square_nums_anno(nums[1:])
21
```

Note: for a variety of technical reasons, these Python type checkers can never be 100% accurate, but in practice they're highly effective!

To enable Python type checking in VSCode, ensure that your Python extension is installed + active, and that it also installed Pylance (it does by default). Then, set: `python.analysis.typeCheckingMode = 'standard'` [\[link\]](#)

23.py 3 Settings

python.analysis.typeCheckingMode 2 Settings Found

User Workspace Backup and Sync

Extensions (2)
Pylance (2)

Python > Analysis: Type Checking Mode

Type checking modes Basic, Standard, and Strict :

Feature	Basic	Standard	Strict
Variable type mismatches	✓	✓	✓
Function return type checks	✓	✓	✓
Type narrowing enforcement		✓	✓
Checking of <code>Any</code> type		✓	✓
Private/protected access checks		✓	✓
Enforces stricter generics usage		✓	✓
Reports missing type annotations			✓
Disallows <code>Any</code> type usage			✓
Requires strict type compatibility			✓
Enforces complete type coverage			✓

For more details, check the [Pyright documentation](#).

standard

(For fun) Programming language type systems

- Every programming language has a type system that defines rules on things like types of variables
- **Static typing**: variable types are defined in the code (often explicitly by the programmer), aka "**at compile time**"
 - Guarantees "type safety" of code, eliminating many kinds of bugs, at the possible cost of dev convenience/velocity
- **Dynamic typing**: variable types are determined at **runtime**
 - More flexible, but more runtime errors

Static typing (Ex: C/C++, Java)

```
int some_fn(int x, float y) {  
    int tmp = 2;  
    if (y == 3.14) {  
        return x * tmp;  
    }  
    else {  
        return x;  
    }  
}
```

```
// In C/C++, this code will NOT run, as it fails  
// the compiler type checker  
int out = some_fn("hi", 3.14);
```

 **Compile Error: type string is not type int**

Dynamic typing (Ex: Python)

```
def some_fn(x, y):  
    tmp = 2  
    if y == 3.14:  
        return x * tmp  
    else:  
        return x
```

```
# Python lets me pass in anything for `x,y`  
# The code will still run!  
>>> some_fn('hi', 3.14)  
hihi
```

(For fun) Programming language type systems

- Python does not enforce type annotations: they are metadata that are ignored during runtime

```
def some_fn(x, y):  
    tmp = 2  
    if y == 3.14:  
        return x * tmp  
    else:  
        return x
```

```
# Python lets me pass in anything for `x,y`  
# The code will still run!  
>>> some_fn('hi', 3.14)  
hihi
```

Instead, third-party developer tools like Pylance [\[link\]](#) and mypy [\[link\]](#) use the type annotations to perform type checking, often integrated with an IDE like VSCode, PyCharm, etc.
Even though the type annotations aren't enforced, it's still a useful signal for developers to use, especially on large codebases.

```
def some_fn_anno(x: int, y: float) -> int:  
    tmp = 2  
    if y == 3.14:  
        return x * tmp  
    else:  
        return x
```

```
# Despite violating the type annotations,  
# the code will still run!  
>>> some_fn_anno('hi', 3.14)  
hihi
```

Aside: some people may feel that treating Python as a statically typed language (via type annotations) violates the spirit of Python's dynamic ("duck") typing.
Personally: I feel that type annotations helps make code easier to understand and maintain. I've embraced it in both my professional and personal Python projects, and over the past few years I see them more regularly in production-quality Python code.