Designing Functions

Announcements

SQL

Conceptual Order of Execution

SQL simple "Order of Execution"

```
SELECT S              3
FROM R1, R2, …        1
WHERE C1              2
```

A **SELECT FROM WHERE (SFW)** query is the most standard SQL query structure that extract records from a relation.

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

"SFW" Examples

SELECT S
FROM R1, R2, …
WHERE C1

3
1
2

A **SELECT FROM WHERE (SFW)** query is the most standard SQL query structure that extract records from a relation.

```
SELECT flavor, price,
price * 2 as tariff_price
FROM cones;
```

Select some columns, possibly apply transformations to the data.

```
SELECT *
FROM cones
WHERE price > 4;
```

Return just the ice cream cones which have a price > $4.00

SQL "Order of Execution"

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

Aggregations happen after filtering.

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**: A1, A2, ... 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

(Review-ish) Python Operations as Data Transformations

Function	Action	Input arguments	Input Fn. Returns	Output
`map`	Transform every item	1 (each item)	"Anything", a new item	**List**: same length, but possibly new values
`filter`	Return a list with fewer items	1 (each item)	A Boolean	**List:** possibly fewer items, values are the same
`reduce`	"Combine" items together	2 (current item, and the previous result)	Type should match the type each item	A "single" item

Python Operations as Data Transformations

Function	SQL satement	SQL Data	“Paraments”	Output
`map`	SELECT	values in a row	Functions and column selectors	**A row** with valued modified, e.g. adding, selecting etc.
`filter`	WHERE	values in a row	Boolean operations based on columns	**The row** or possibly no rows.
`reduce`	GROUP BY	A table, multiple rows	Column selectors — group by matches on equality.	A set of groups.

Joins Practice

Returning to Ice Cream Cones

How many dollars (“Total Sales”) did each salesperson sell?

JOINing on cone id

cones:

id	flavor	color	price
1	strawberry	pink	3.5
2	chocolate	brown	4.75

sales:

id	cashier	cone_id
1	Baskin	2
10	Jerry	6

```
SELECT Cashier, SUM(pirce)___ AS “Total Sales”
  FROM _____cones c, sales s_
  WHERE _s.cone_id = c.id_____
  GROUP BY cashier;
```

Cashier	Total Sales
Ben	$9.75
Jerry	…
Baskin	…

Discussion Question

What's the maximum difference between leg count for two animals with the same weight?

Approach #1: Consider all pairs of animals.

```
SELECT MAX(a.legs - b.legs) AS difference
  FROM animals AS a, animals AS b
 WHERE a.weight = b.weight;
```

Approach #2: Group by weight.

```
SELECT MAX(legs) - MIN(legs) AS difference
  FROM animals
  GROUP BY weight
  ORDER BY difference DESC
  LIMIT 1;
```

animals:

kind	legs	weight
dog	4	20
cat	4	10
ferret	4	10
parrot	2	6
penguin	2	10
t-rex	2	12000

difference
2

Discussion Question

What are all the kinds of animals that have the maximal number of legs?

```
sqlite> SELECT * FROM animals WHERE legs = MAX(legs);
Parse error: misuse of aggregate function MAX()
```

animals:

kind	legs	weight
dog	4	20
cat	4	10
ferret	4	10
parrot	2	6
penguin	2	10
t-rex	2	12000

Approach #1: Give the maximum number of legs a name.

```
CREATE TABLE m AS SELECT MAX(legs) AS max_legs FROM animals;
SELECT kind FROM animals, m WHERE legs = max_legs;
```

Approach #2: For each kind of animal, compare its legs to the maximum legs by grouping.

```
SELECT a.kind FROM animals AS a, animals AS b GROUP BY a.kind HAVING a.legs = MAX(b.legs);
```

Implementing Functions

A Slight Variant of Fall 2022 Midterm 1 3(b)

Implement nearest_prime, which takes an integer n above 5. It returns the nearest prime number to n. If two prime numbers are equally close to n, return the larger one. Assume is_prime(n) is implemented already.

```
def nearest_prime(n):
    """Return the nearest prime number to n.
    In a tie, return the larger one.

    >>> nearest_prime(8)
    7
    >>> nearest_prime(11)
    11
    >>> nearest_prime(21)
    23
    """
    k = 0
    while True:
        if _____is_prime(23)_____:
            return __23___
        if _______:
            k = -k
        else:
            k = ________
```

Example: n is 21

keep looking for a prime

From discussion:

Describe a process (in English) that computes the output from the input using simple steps.

Figure out what additional names you'll need to carry out this process.

Implement the process in code using those additional names.

Read the description

Verify the examples & pick a simple one

Read the template

Annotate names with values from your chosen example

Write code to compute the result

Did you really return the right thing?

Check your solution with the other examples

A Slight Variant of Fall 2022 Midterm 1 3(b)

Implement nearest_prime, which takes an integer n above 5. It returns the nearest prime number to n. If two prime numbers are equally close to n, return the larger one. Assume is_prime(n) is implemented already.

```
def nearest_prime(n):
    """Return the nearest prime number to n.
    In a tie, return the larger one.

    >>> nearest_prime(8)
    7
    >>> nearest_prime(11)
    11
    >>> nearest_prime(21)
    23
    """
    k = 0
    while True:
        if is_prime(n + k):
            return n + k
        if k > 0:
            k = -k
        else:
            k = -k + 1
```

Example: n is 21

is_prime(23)
23

keep looking for a prime

From discussion:

Describe a process (in English) that computes the output from the input using simple steps.

Figure out what additional names you'll need to carry out this process.

Implement the process in code using those additional names.

Process:
Check whether a number is prime in this order:
- original n
- n + 1
- n – 1
- n + 2
- n – 2
- n + 3
- n – 3
- n + 4
...

All of these look like n + k for various k

(Demo)

Designing Functions

How to Design Programs

From Problem Analysis to Data Definitions
Identify the information that must be represented and how it is represented in the chosen programming language. Formulate data definitions and illustrate them with examples.

Signature, Purpose Statement, Header
State what kind of data the desired function consumes and produces. Formulate a concise answer to the question *what* the function computes. Define a stub that lives up to the signature.

Functional Examples
Work through examples that illustrate the function's purpose.

Function Template
Translate the data definitions into an outline of the function.

Function Definition
Fill in the gaps in the function template. Exploit the purpose statement and the examples.

Testing
Articulate the examples as tests and ensure that the function passes all. Doing so discovers mistakes. Tests also supplement examples in that they help others read and understand the definition when the need arises—and it will arise for any serious program.

https://htdp.org/2018-01-06/Book/

Tree Processing

Tree-Structured Data

```
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = list(branches)

    def is_leaf(self):
        return not self.branches
```

A tree can contains other trees:

```
[5, [6, 7], 8, [[9], 10]]
```

```
(+ 5 (- 6 7) 8 (* (- 9) 10))
```

```
(S
  (NP (JJ Short) (NNS cuts))
  (VP (VBP make)
      (NP (JJ long) (NNS delays)))
  (. .))
```

```
<ul>
  <li>Midterm <b>1</b></li>
  <li>Midterm <b>2</b></li>
</ul>
```

Tree processing often involves recursive calls on subtrees

Designing a Function

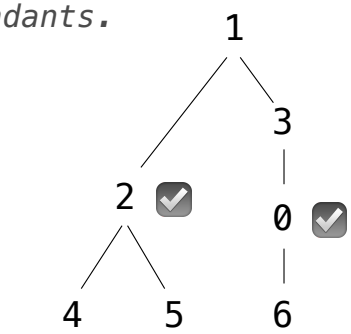
Implement **smalls**, which takes a Tree instance t containing integer labels. It returns the non-leaf nodes in t whose labels are smaller than any labels of their descendant nodes.

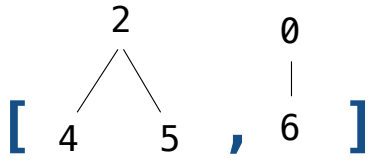
```
def smalls(t):
    """Return a list of the non-leaf nodes in t that are smaller than all their descendants.

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]

    """
    result = []
    def process(t):
        if t.is_leaf():
            return t.label
        else:



            return min(...)
    process(t)
    return result
```

Signature: Tree -> List of Trees

Signature: Tree -> number

"Find smallest label in t & maybe add t to result"





Designing a Function

Implement **smalls**, which takes a Tree instance t containing integer labels. It returns the non-leaf nodes in t whose labels are smaller than any labels of their descendant nodes.

```
def smalls(t):
    """Return a list of the non-leaf nodes in t that are smaller than all descendants.

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]

    """
    result = []
    def process(t):
        if t.is_leaf():
            return ____________________________________________
        else:
            smallest = ____________________________________________
            if ____________________________________________:
                ____________________________________________
            return min(smallest, t.label)
    process(t)
    return result
```

Signature: Tree -> List of Trees

Signature: Tree -> number

"Find smallest label in t & maybe add t to result"

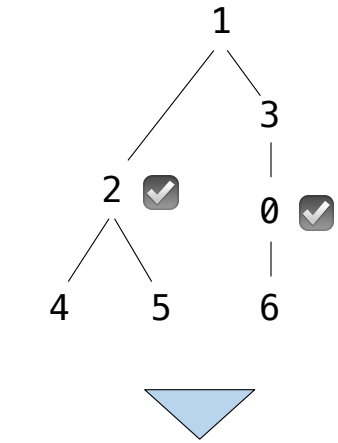
t.label

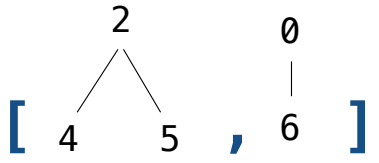
min([process(b) for b in t.branches])

t.label < smallest

result.append(t)

smallest label in a branch of t





Fall 2022 Midterm 2 Question 4(b)

A *hydra* is a Tree with a special structure. Each node has 0 or 2 children. All leaves are heads labeled 1. Each non-leaf body node is labeled with the number of leaves among its descendants.

Implement **chop_head(hydra, n)**, which takes a hydra and a positive integer n. It mutates the hydra by replacing the nth head from the left with two new adjacent heads & updating all ancestor labels.

```
def chop_head(hydra,  n):
    assert n > 0 and n <= hydra.label
    if hydra.is_leaf():
        hydra.label = 2
        hydra.branches = [Tree(1),  Tree(1)]
    else:
        hydra.label += 1
        left, right = hydra.branches
        if n > left.label:
            chop_head(right, n - left.label)
        else:
            chop_head(left, n)
```

Signature: (hydra, int) -> None

Mutate the hydra

Update ancestor

nth head