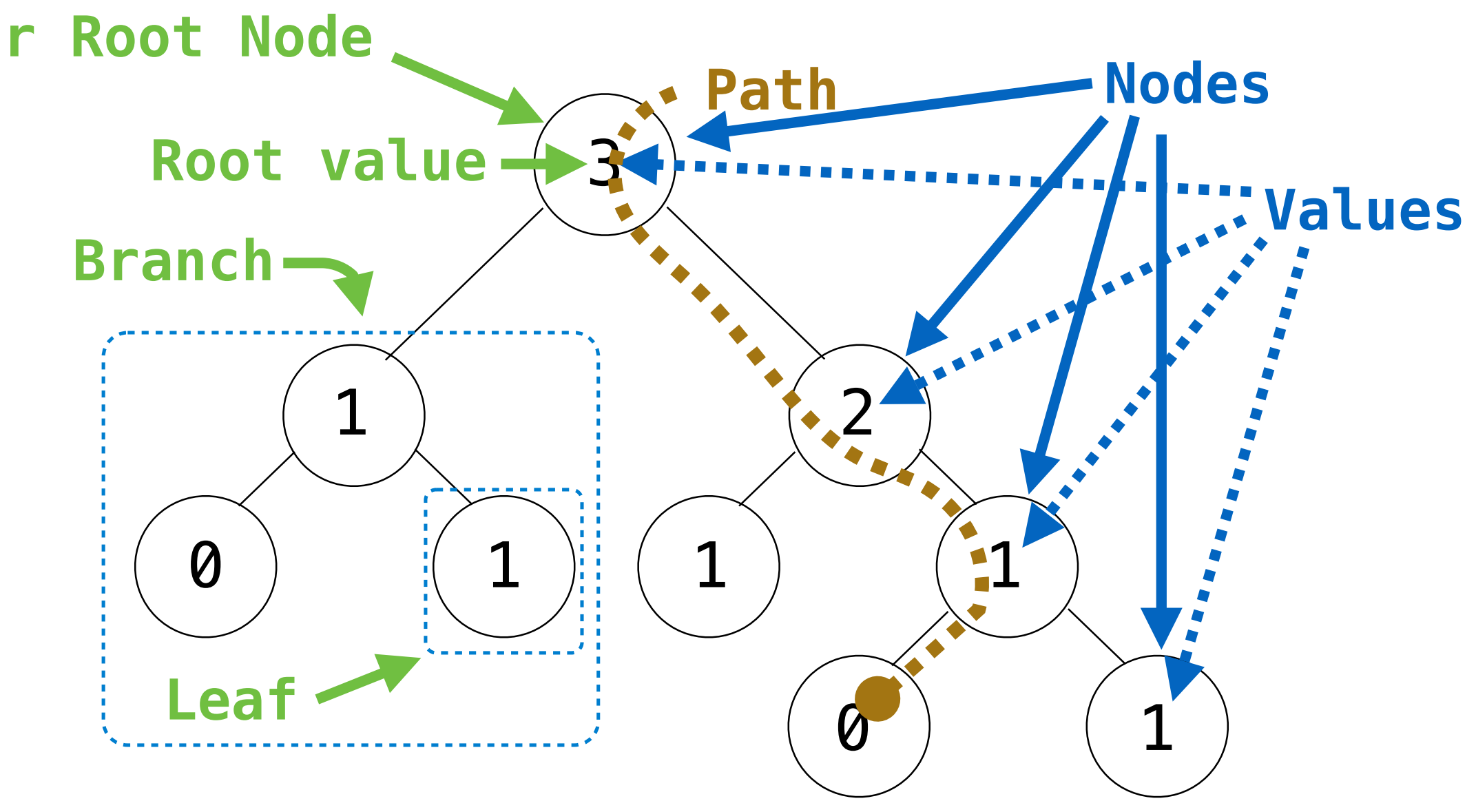
DATA C88C Final Reference Sheet Page 1 of 2

Recursive description:

- A **tree** has a root **value** and a list of **branches**
- Each branch is a **tree**
- A tree with zero branches is called a **leaf**

Relative description:

- Each location in a tree is a **node**
- Each **node** has a **value**
- One node can be the **parent/child** of another

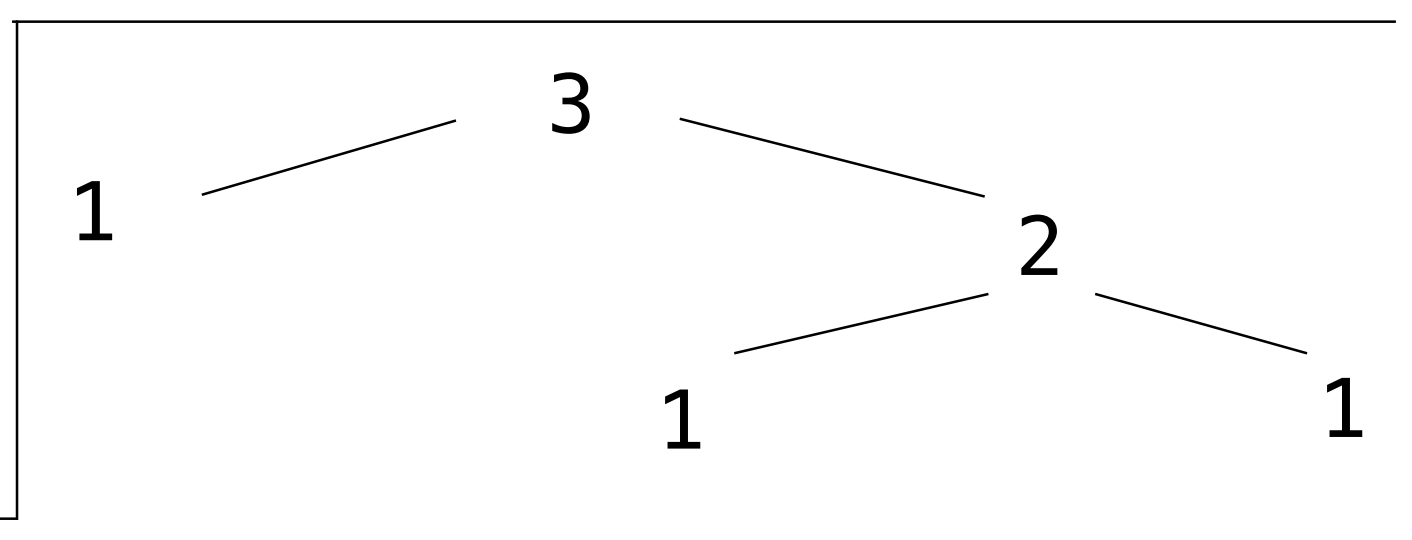


```
class Tree:
    def __init__(self, value, branches=[]):
        self.value = value
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)

    def is_leaf(self):
        return not self.branches

def leaves(tree):
   """The leaf values in a tree."""
   if tree.is_leaf():
      return [tree.value]
    else:
      lst = []
      for b in tree.branches:
          lst.extend(leaves(b))
      return lst
```

Built-in isinstance function: returns True if branch has a class that *is* **or** *inherits from* Tree



```
Tree(3,
    [Tree(1), Tree(2,
        [Tree(1), Tree(1)])])

def fib_tree(n):
    if n == 0 or n == 1:
        return Tree(n)
    else:
        left = fib_tree(n - 2)
        right = fib_tree(n - 1)
        fib_n = left.value + right.value
        return Tree(fib_n, [left, right])
```

Exceptions are raised with a raise statement.

`raise <expr>`

<expr> must evaluate to a subclass of BaseException or an instance of one.

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
```

The <try suite> is executed first.

If, during the course of executing the <try suite>, an exception is raised that is not handled otherwise, and

```
>>> try:
        x = 1 / 0
    except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0

handling a <class 'ZeroDivisionError'>
>>> x
0
```

If the class of the exception inherits from <exception class>, then

The <except suite> is executed, with <name> bound to the exception.

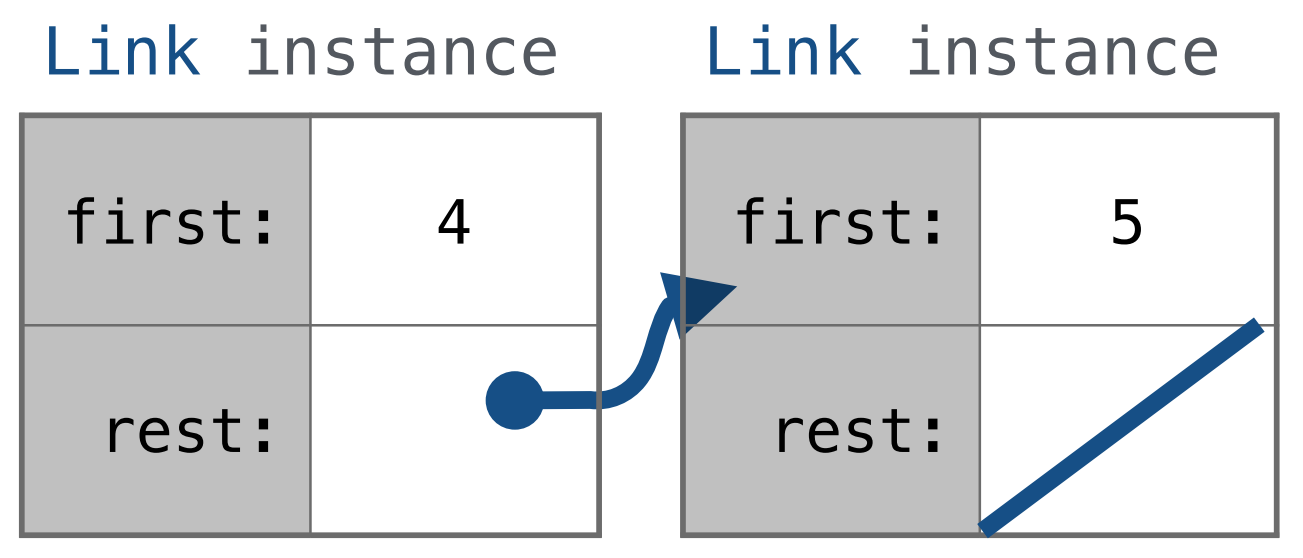
```
class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or
            isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest:
            rest = ', ' + repr(self.rest)
        else:
            rest = ''
        return 'Link(' + repr(self.first) + rest + ')'

    def __str__(self):
        string = '('
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + ')'
```

Some zero length sequence



```
>>> s = Link(4, Link(5))
>>> s
Link(4, Link(5))
>>> s.first
4
>>> s.rest
Link(5)
>>> print(s)
(4 5)
>>> print(s.rest)
(5)
>>> s.rest.rest is
Link.empty
True
```

Anatomy of a recursive function:

- The **def statement header** is like any function
- Conditional statements check for **base cases**
- Base cases are evaluated **without recursive calls**
- Recursive cases are evaluated **with recursive calls**

```
def sum_digits(n):
  "Sum the digits of positive integer n."
  if n < 10:
      return n
  else:
      all_but_last, last = n // 10, n % 10
      return sum_digits(all_but_last) + last
```

- **Recursive decomposition:** finding simpler instances of a problem.
- E.g., count_partitions(6, 4)
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - count_partitions(2, 4)
  - count_partitions(6, 3)
- Tree recursion often involves exploring different choices.

```
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```

Python object system:

Idea: All bank accounts have a **balance** and an account **holder**; the **Account** class should add those attributes to each of its instances

A new instance is created by calling a class

```
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

An account instance

balance: 0   holder: 'Jim'

When a class is called:

1. A new instance of that class is created:
2. The `__init__` method of the class is called with the new object as its first argument (named self), along with any additional arguments provided in the call expression.

```
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```

`__init__` is called a constructor

self should always be bound to an instance of the Account class or a subclass of Account

Function call: all arguments within parentheses

Method invocation: One object before the dot and other arguments within parentheses

```
>>> type(Account.deposit)
<class 'function'>
>>> type(a.deposit)
<class 'method'>

>>> Account.deposit(a, 5)
10
>>> a.deposit(2)
12
```

Call expression

Dot expression

`<expression> . <name>`

The <expression> can be any valid Python expression.
The <name> must be a simple name.
Evaluates to the value of the attribute looked up by <name> in the object that is the value of the <expression>.

To evaluate a dot expression:

1. Evaluate the <expression> to the left of the dot, which yields the object of the dot expression
2. <name> is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned
3. If not, <name> is looked up in the class, which yields a class attribute value
4. That value is returned unless it is a function, in which case a bound method is returned instead

Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression

- If the object is an instance, then assignment sets an instance attribute
- If the object is a class, then assignment sets a class attribute

Account class attributes

interest: ~~0.02~~ ~~0.04~~ 0.05
(withdraw, deposit, `__init__`)

Instance attributes of jim_account

balance: 0
holder: 'Jim'
interest: 0.08

Instance attributes of tom_account

balance: 0
holder: 'Tom'

```
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
>>> jim_account.interest
0.04
```

```
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
>>> jim_account.interest
0.08
```

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
                                   or
        return super().withdraw(      amount + self.withdraw_fee)
```

To look up a name in a class:

1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

```
>>> ch = CheckingAccount('Tom')  # Calls Account.__init__
>>> ch.interest     # Found in CheckingAccount
0.01
>>> ch.deposit(20)  # Found in Account
20
>>> ch.withdraw(5)  # Found in CheckingAccount
14
```

DATA C88C Final Reference Sheet Page 2 of 2

```
iter(iterable):               >>> s = [3, 4, 5]  >>> d = {'one': 1, 'two': 2, 'three': 3}
  Return an iterator          >>> t = iter(s)    >>> k = iter(d)
  over the elements of        >>> next(t)        >>> next(k)      >>> v = iter(d.values())
  an iterable value           3                  'one'            >>> next(v)
next(iterator):               >>> next(t)        >>> next(k)      1
  Return the next element     4                  'two'            >>> next(v)
                                                                  2
```

---

*A generator function is a function that **yield**s values instead of **return**ing.*

```
>>> def plus_minus(x):  >>> t = plus_minus(3)  def a_then_b(a, b):
...     yield x         >>> next(t)                yield from a
...     yield -x        3                          yield from b
                        >>> next(t)            >>> list(a_then_b([3, 4], [5, 6]))
                        -3                     [3, 4, 5, 6]
```

Efficiency

Constant growth. E.g., accessing a value from a dictionary. Increasing *n* doesn't affect time $O(1)$

Logarithmic growth. E.g., binary search Doubling *n* only increments *time* by a constant $O(\log n)$

Linear growth. E.g., iterating over a list of length n Incrementing *n* increases *time* by a constant $O(n)$

Quadratic growth. E.g., finding all pairs of a list of integers (double for loop) $O(n^2)$

Incrementing *n* increases *time* by *n* times a constant

Exponential growth. E.g., recursive fib $O(b^n)$

Incrementing *n* multiplies *time* by a constant

```
def perms(lst):
    """Generates the permutations of lst one by one.
    >>> perms = perms([1, 2, 3])
    >>> p = list(perms)
    >>> p.sort()
    >>> p [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
    """
    if lst == []:
        yield []
    else:
        for perm in perms(lst[1:]):
            for i in range(len(lst)):
                yield perm[:i] + [lst[0]] + perm[i:]
```

A **table** has columns and rows

Latitude	Longitude	Name
38	122	Berkeley
42	71	Cambridge
45	93	Minneapolis

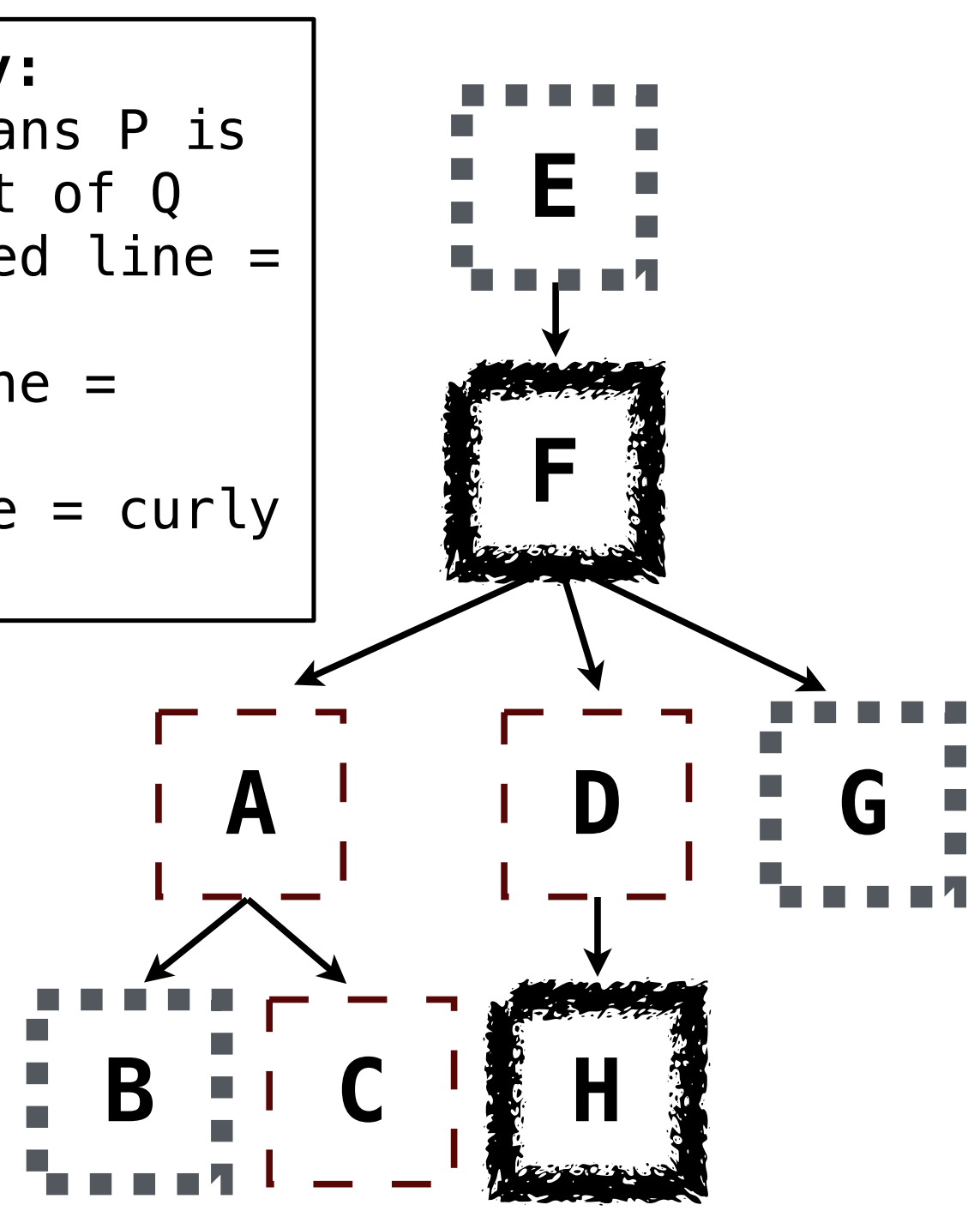
A **column** has a name and a type

A **row** has a value for each column

```
SELECT [expression] AS [name], [expression] AS [name], ... ;
SELECT [columns] FROM [table] WHERE [condition] ORDER BY [order];
```

```
CREATE TABLE parents AS
  SELECT "D" AS parent, "H" AS child UNION
  SELECT "A"          , "B"          UNION
  SELECT "A"          , "C"          UNION
  SELECT "F"          , "A"          UNION
  SELECT "F"          , "D"          UNION
  SELECT "F"          , "G"          UNION
  SELECT "E"          , "F";
```

Diagram Key:
- P -> Q means P is the parent of Q
- Thin dashed line = long fur
- Dotted line = short fur
- Thick line = curly fur



```
CREATE TABLE dogs AS
  SELECT "A" AS name, "long" AS fur UNION
  SELECT "B"        , "short"       UNION
  SELECT "C"        , "long"        UNION
  SELECT "D"        , "long"        UNION
  SELECT "E"        , "short"       UNION
  SELECT "F"        , "curly"       UNION
  SELECT "G"        , "short"       UNION
  SELECT "H"        , "curly";
```

```
SELECT a.child AS first, b.child AS second
  FROM parents AS a, parents AS b
  WHERE a.parent = b.parent AND a.child < b.child;
```

first	second
B	C
A	D
A	G
D	G

String values can be combined to form longer strings

```
sqlite> SELECT "hello," || " world";
hello, world
```

Basic string manipulation is built into SQL, but differs from Python

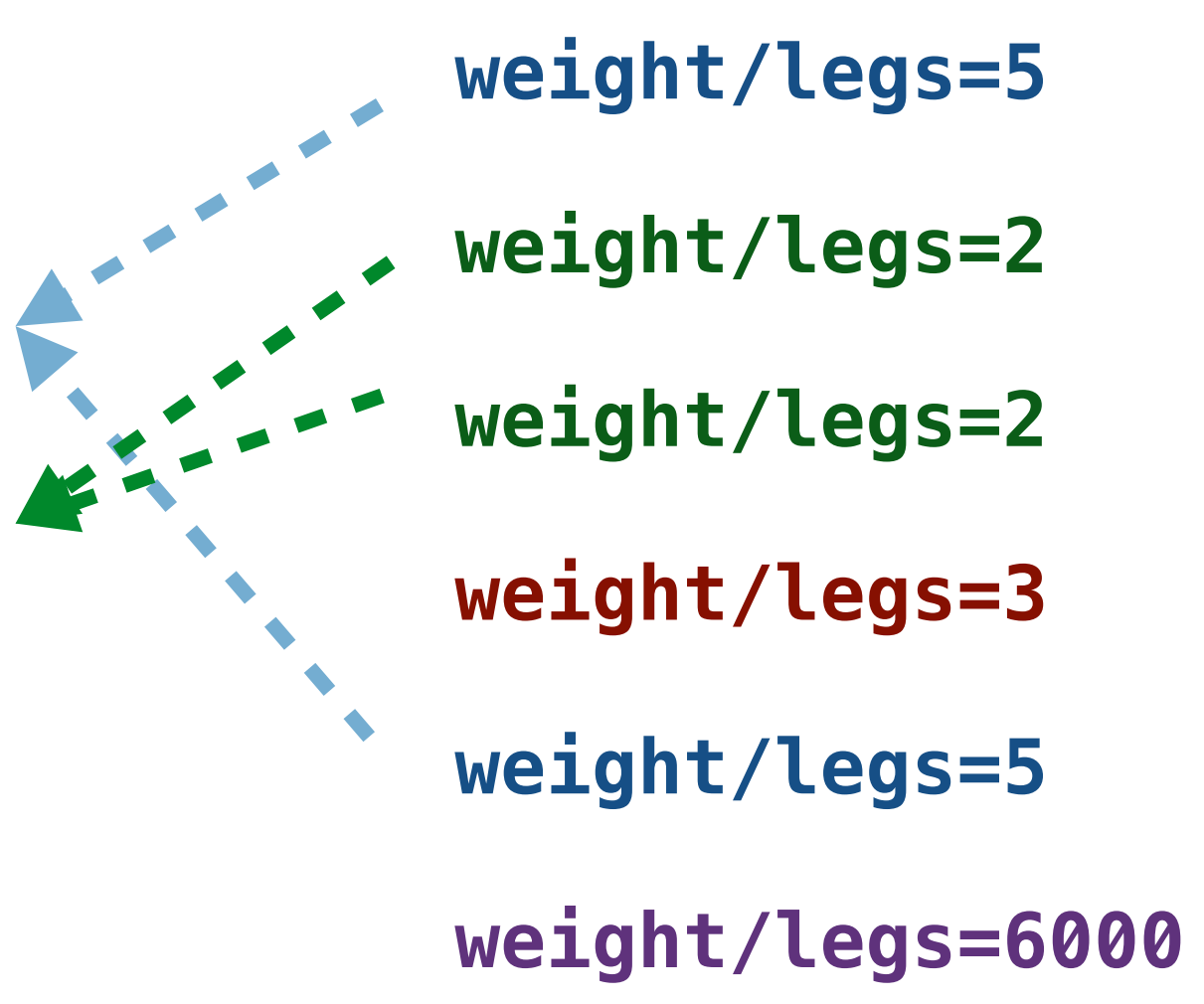
```
sqlite> CREATE TABLE phrase AS SELECT "hello, world" AS s;
sqlite> SELECT substr(s, 4, 2) || substr(s, instr(s, " ") + 1, 1)
          FROM phrase;
low
```

The number of groups is the number of unique values of an expression

A HAVING clause filters the set of groups that are aggregated

```
SELECT weight / legs, COUNT(*) FROM animals
                               GROUP BY weight / legs
                               HAVING COUNT(*) > 1;
```

weight/ legs	COUNT(*)
5	2
2	2



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

An aggregate function in the [columns] clause computes a value from a group of rows:

- MAX([expression]) evaluates to the largest value of [expression] for any row in a group
- COUNT(*) evaluates to the number of rows in a group
- MIN, SUM, & AVG are also aggregate functions similar to MAX

With no GROUP BY clause, aggregation is performed over all rows:

```
SELECT MAX(legs) FROM animals;
```

MAX(legs)
4