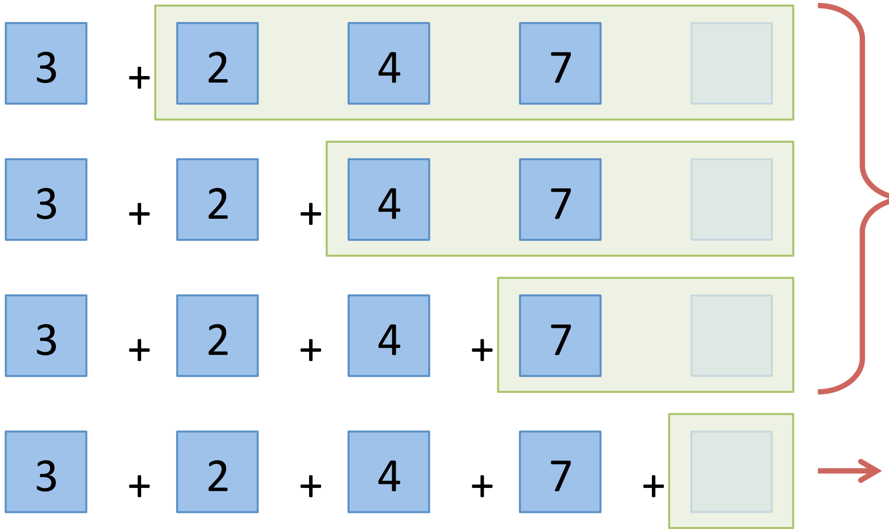
Intro to Computer Science

Topic	Lab	Book
Recursion	13—14	12
Objects	15—17	10—11
SQL		
Web stuff		

Recursive addition



The head of a list
“plus” the tail

Empty lists are zero

```
def rsum(h):
    if not h:
        return 0
    else:
        return h[0] + rsum(h[1:])
```

Factorial: the recursive approach

$$\left.\begin{aligned} 5! &= 5\times 4! \\ &= 5\times 4\times 3! \\ &= 5\times 4\times 3\times 2! \\ &= 5\times 4\times 3\times 2\times 1! \end{aligned}\right\}\text{Getting closer a base case...}$$

$$= 5\times 4\times 3\times 2\times 1\times 0! \longrightarrow \text{... base case!}$$

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Backtracking

The idea

- From the start of the board, there are several potential solutions
- The naïve approach is to try every one, from the start of the board
- Backtracking allows us to incrementally try potentials

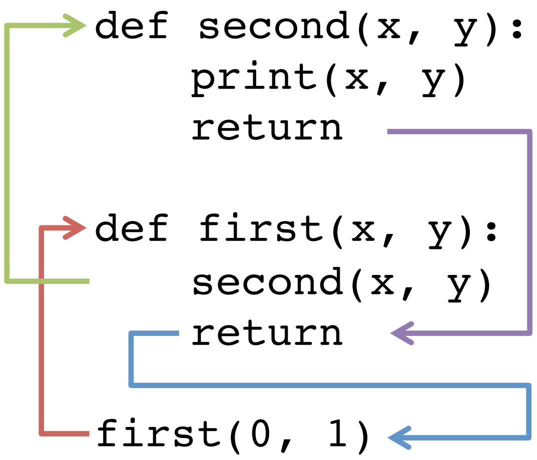
Algorithm

Choice	Logic
Accept	have we solved the problem
Reject	can we say that this is definitely not a solution
Move	step toward a potential solution
recurse	

Calling a function: what really happens

- Parameters, local variables, and return locations all require memory
- This allocated memory is called a *stack frame*
  - Parameters
  - Local variables
  - Return location

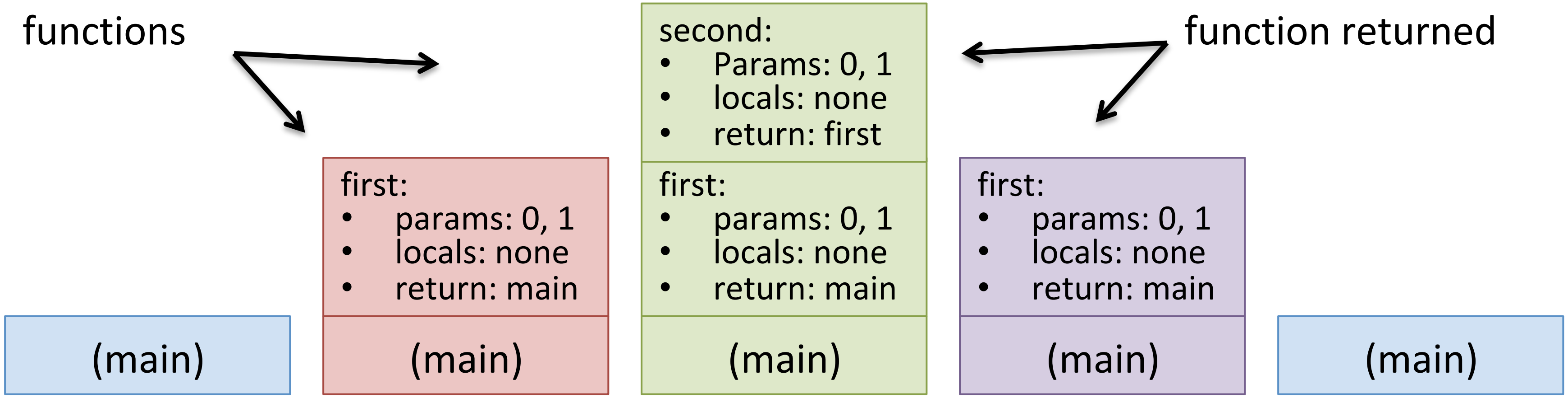
Calling a function: what really happens



```
def second(x, y):
    print(x, y)
    return

def first(x, y):
    second(x, y)
    return

first(0, 1)
```

Memory is allocated as we call more functions

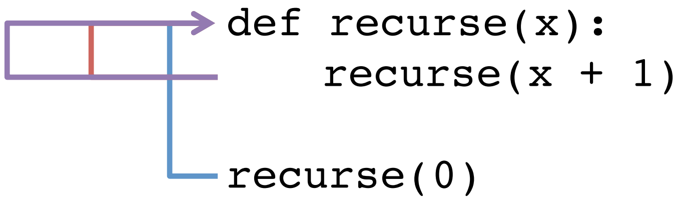
Memory is de-allocated when the function returned



second:
- Params: 0, 1
- locals: none
- return: first

first:
- params: 0, 1
- locals: none
- return: main

Make it stop!



```
def recurse(x):
    recurse(x + 1)

recurse(0)
```

…

recurse:
- params: 2
- locals: none
- return: main

recurse:
- params: 1
- locals: none
- return: main

recurse:
- params: 0
- locals: none
- return: main

(main)

Classes are made up of

Attributes

- *Variables* contained within the class
- Accessed using ‘self’

Methods

- *Functions* contained within the class
- Accessed using ‘self’
- Can utilize
  - Parameters of the function
  - Other other attributes

Look like any other method… almost

```
class Singer():
   def note(self):
      return ‘a minor’
```

- Standard class definition
- Standard function definition
  - That’s in the class definition!
  - Whose first parameter is `self`

```
s = Singer()
print(s.note())
```

- Instantiate the class
- Call the `note` method within the instance

Instantiation: the constructor

You call

```
s = Singer()
```

Python does

```
class Singer:
   def __init__(self):
      return
```

What to notice:

1. Function name is special
   - Cannot change this!
2. Double underscore syntax
3. Obligatory `self` parameter

Instantiation: attributes

You call

```
bieb = Singer('Bieber')
print(bieb.name)

mj = Singer('MJ')
print(mj.name)
```

Python does

```
class Singer:
    def __init__(self, name):
        self.name = name
        return
```

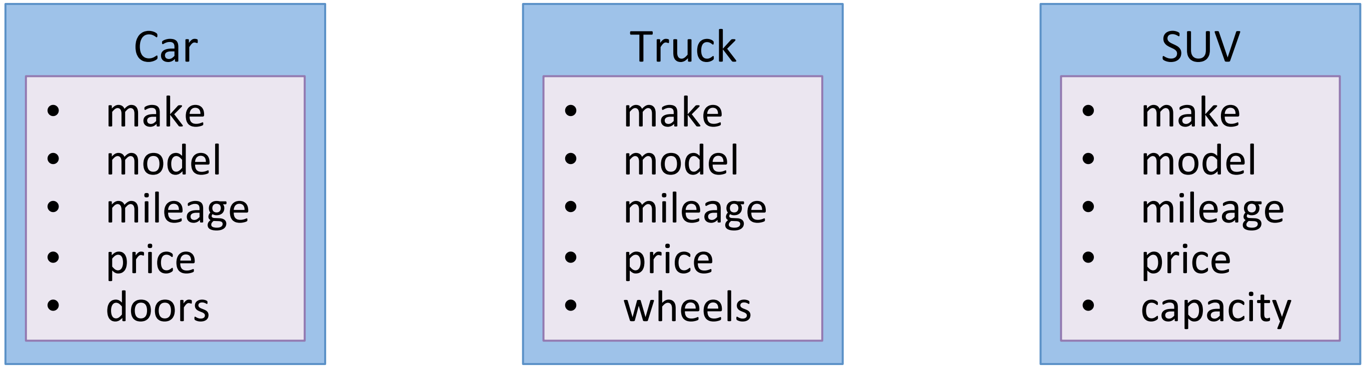
- `name` is now an additional variable in the `Singer` class
    - It can be used throughout the class, akin to a global variable within the class
- This assignment happens during instantiation
- Each particular instance will have the value that is specified

Inheritance

- One way to save work is to *specialize* things that have already been built:
  - An ordering of objects that are related, more increasingly specific

Solution 1 (not good!)

- Make an object for each
- Include five attributes for each



Car

- make
- model
- mileage
- price
- doors

Truck

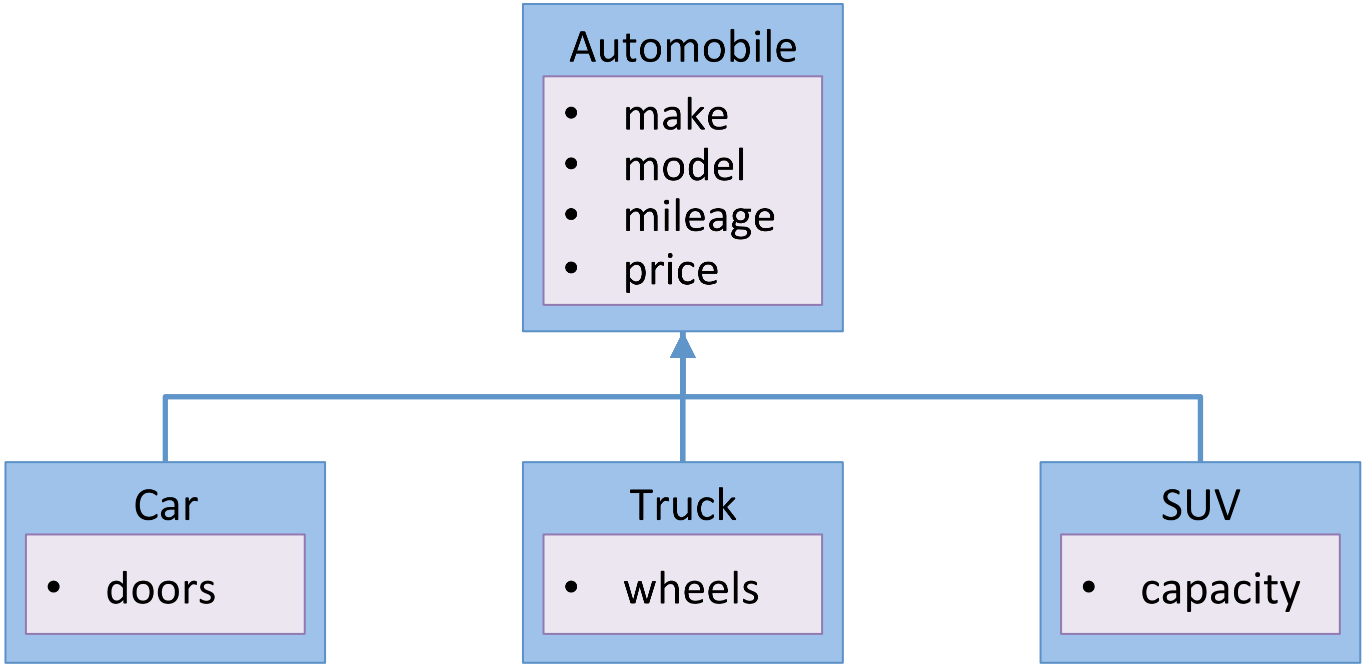
- make
- model
- mileage
- price
- wheels

SUV

- make
- model
- mileage
- price
- capacity

Solution 2 (good!)

- Notice that they only differ by a one attribute
  - Build a parent class containing the common attributes
  - Have specialized classes for the differing attribute



Instantiating a child

- If no constructor is present, the parents constructor will be called
- Otherwise, the child can explicitly call the parent constructor if it chooses

Privacy please

- By default, everything that is defined in the class is “available” (visible) to instances of that class
  - Both methods and attributes
- Private object members are only “visible” from within the class
  - Only by `self`
  - Not even visible by subclasses!

HOWTO: The underscore

```
class Singer:
   __init__(self, name):
      self.name = name
      self.__last_night = ‘wild’
```

Indicates the attribute is private

```
bieb = Singer(‘Bieber’)

print(bieb.name)                ☺

print(bieb.__last_night)        ☹
```

The underscore

```
class Singer:
    __init__(self, name):
        self.name = name
        self.__last_night = ‘wild’

    def about_last_night():
        if self.__last_night == ‘appropriate’:
            return self.__last_night
        else:
            return ‘nothing’
```

Internally, we’re able to see and use private information

The underscore

```
class Singer:
    __init__(self, name):
        self.name = name
        self.__last_night = ‘wild’

    def __reveal_last_night(self):
        return False
```

Aside from the special (internal) methods, class authors can define their own privates methods as well

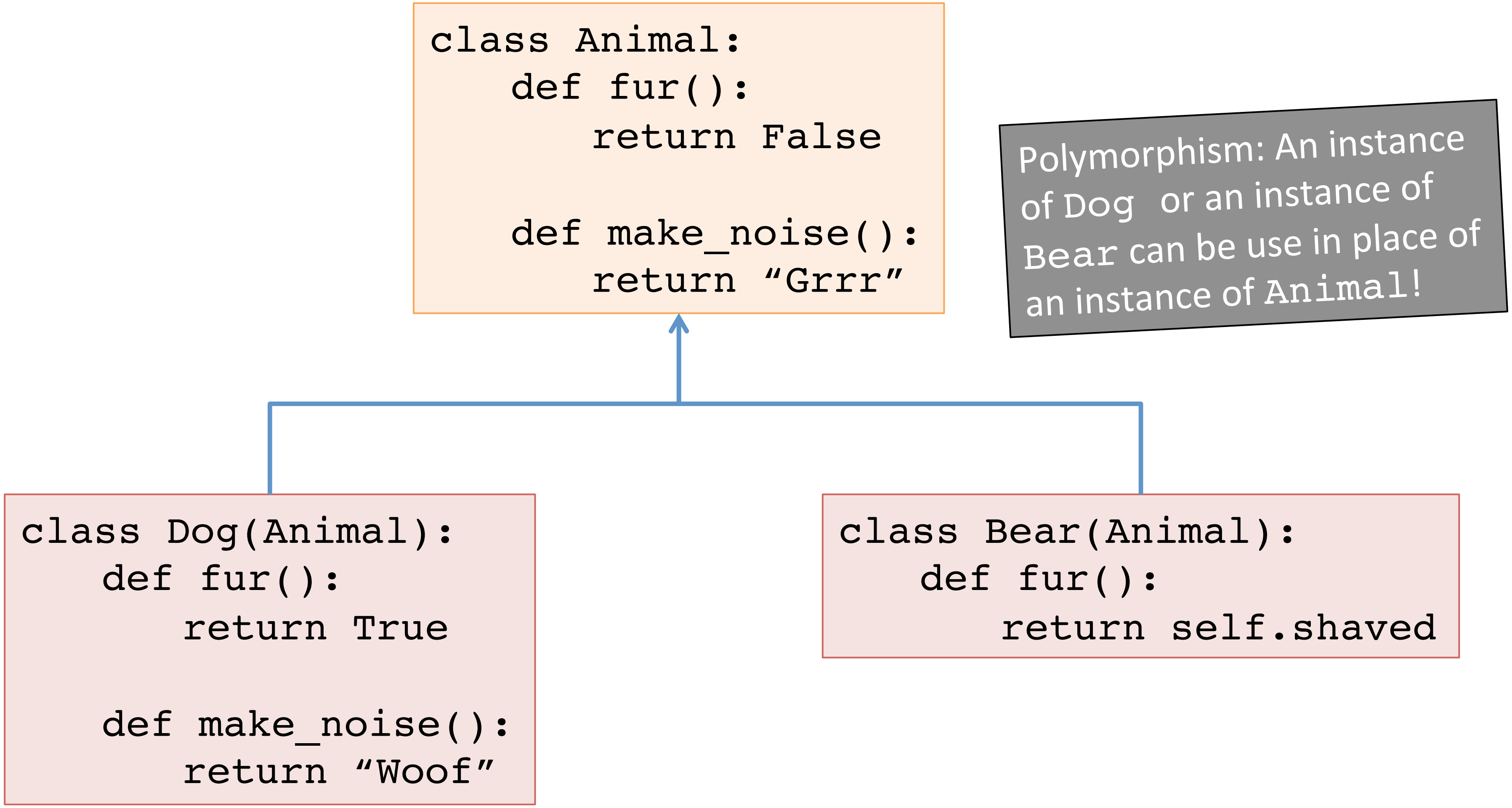
```
bieb = Singer(‘Bieber’)

print(bieb.__last_night)   ☹

bieb.__reveal_last_night() ☹ ☹
```

Polymorphism

This allows us to do something cool:

- Recall that a sub-class contains all of the functionality of its respective super-class
- We can use this to make “generic” programs
  - Known as *polymorphism*: having more than one form
- Essentially: using a class through its parent interface

Example: animals



```
class Animal:
    def fur():
        return False

    def make_noise():
        return “Grrr”
```

```
class Dog(Animal):
    def fur():
        return True

    def make_noise():
        return “Woof”
```

```
class Bear(Animal):
    def fur():
        return self.shaved
```

Polymorphism: An instance of `Dog` or an instance of `Bear` can be use in place of an instance of `Animal`!

In SQL

- SELECT [attributes]
  - SELECT uid, name, major
  - SELECT *
- FROM [relations]
  - FROM students
- WHERE [predicates]
  - WHERE uid > 2 AND major = ‘CS’

- Predicate gotchas
  - Not equal is ‘<>’
  - Special comparisons for NULL values
    - a NOT NULL
    - a IS NULL
  - String values go between single quotes (‘ ’), not double quotes (“ ”)

Joins

Type	Names	
Cross join	• Cartesian product	Every tuple from R_1 combined with ever tuple from R_2
Inner join	• Natural join • Equi-join	Tuples in R_1 and R_2 with some matching attribute
Outer join	• Left-outer join • Right-outer join • Full-outer join	Tuples from one relation that may not have a matching attribute with the other

- We will mostly focus on equi-joins

Cross join

- The *cross join* combines two or more relations
  - Sometimes referred to as the *Cartesian product*

UID	Name	Major
1	Alex	CS
2	Lisa	Bio
3	Shaun	CS

UID	Name	Major
1	Bob	EE
2	Paula	Chem

Cross join

- Combine every tuple from one set with every tuple from the other
- This can be done with an arbitrary number of relations (tables) and an arbitrary number of tuples (rows)!

UID	Name	Major	PID	Name	Dept
1	Alex	CS	1	Bob	EE
1	Alex	CS	2	Paula	Chem
2	Lisa	Bio	1	Bob	EE
2	Lisa	Bio	2	Paula	Chem
3	Shaun	CS	1	Bob	EE
3	Shaun	CS	2	Paula	Chem

Inner join

- Cross joins give no regard to the whether rows should be combined
  - Inner joins allow us to specify the linking data
- Concerned with tuples that have matching attributes

Natural inner join

- The natural join of R_1 and R_2 is the cross join in which the common attributes in R_1 and R_2 are equal

UID	Name	Dept
1	Alex	CS
2	Lisa	Bio
3	Shaun	EE
4	Hillary	ME

Dept	Head
CS	Lucy
Chem	Diane
Bio	Roger

UID	Name	Dept	Head
1	Alex	CS	Lucy
2	Lisa	Bio	Roger

Equi-join

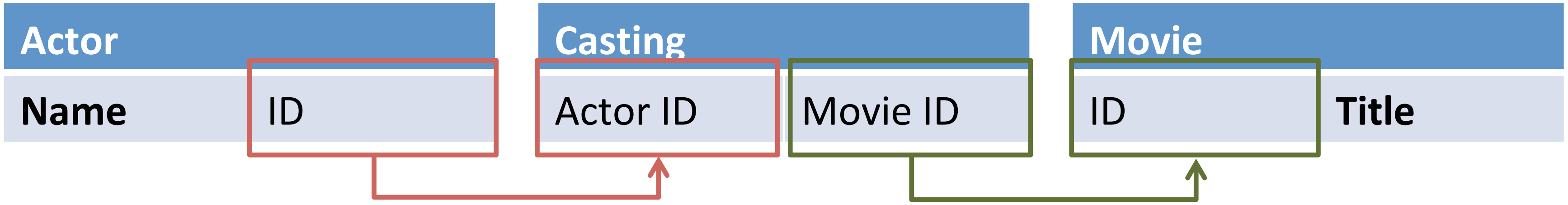
- The natural join of R_1 and R_2 is the cross join in which the specified attributes in R_1 and R_2 are equal

UID	Name	Major
1	Alex	CS
2	Lisa	Bio
3	Shaun	EE
4	Hillary	ME

ID	Name	Dept
3	Lucy	CS
4	Diane	Chem
5	Roger	Bio

UID	Name	Major	Dept	Head
3	Shaun	EE	CS	Lucy
4	Hillary	ME	Chem	Diane

Going to the movies



Actor

Name | ID

Casting

Actor ID | Movie ID

Movie

ID | Title

```
SELECT actor.name, movie.title FROM actor

JOIN casting ON actor.id = casting.actorid

JOIN movie ON casting.movieid = movie.id
```

```
SELECT actor.name, movie.title
FROM actor, casting, movie
WHERE actor.id = casting.actorid AND casting.movieid = moviel.id
```

← Alternative

Outer join

- Concerned with tuples that *do not* having matching similar attributes
  - A much different concept to the inner joins
- We introduce the existence of a NULL value

Left-outer join

- The left-outer join is
  - The natural join of relations R_1 and R_2
  - Tuples in R_1 that have no matching tuple in R_2

UID	Name	Dept
1	Alex	CS
2	Lisa	Bio
3	Shaun	EE
4	Hillary	ME

Name	Dept
Lucy	CS
Diane	Chem
Roger	Bio

UID	Name	Dept	Head
1	Alex	CS	Lucy
2	Lisa	Bio	Roger
3	Shaun	EE	NULL
4	Hillary	ME	NULL

Right-outer join

- The left-outer join is
  - The natural join of relations R_1 and R_2
  - Tuples in R_2 that have no matching tuple in R_1

UID	Name	Dept
1	Alex	CS
2	Lisa	Bio
3	Shaun	EE
4	Hillary	ME

Name	Dept
Lucy	CS
Diane	Chem
Roger	Bio

UID	Name	Dept	Head
1	Alex	CS	Lucy
NULL	NULL	Chem	Diane
2	Lisa	Bio	Roger

HTTP POST

- GET passes information as part of the URL
  - Query string
- POST passes it as part of the message body
  - Along with the URL, send a message containing the form information

Comparison

GET	POST
Information sent via the query string	Information sent within the request
Page creator can manipulate links	Information is exchanged through forms
Limited by URL length (web server dependent)	Effectively no limit on how large the POST can be
Information that is sent is visible to users	Information is “hidden” from view (good for sensitive information)
Page reloading seems normal	Page reload may prompt “confirmation” notification from browser

- Today we will use both just for practice
- In reality, choice will depend on conditions
- *Understanding both is advantageous*