SURVEY: How long did Assignment 10 exercises & WDIDs take?

A) <~ 1 hour

B) ~ 2 hours

C) ~ 3 hours

D) > 3 hours

E) I didn’t do the exercises & WDIDs

SURVEY: How long did Assignment 11 exercises & WDIDs take?

A) <~ 1 hour

B) ~ 2 hours

C) ~ 3 hours

D) > 3 hours

E) I didn’t do the exercises & WDIDs

SURVEY: How long did Assignment 10 homework take?

A) <~ 1 hour

B) ~ 2 hours

C) ~ 3 hours

D) > 3 hours

E) I didn’t do the homework yet

SURVEY: How much work is this class compared to other astro classes? (if you’re not taking any, just don’t answer)

A) A lot more

B) A little more

C) About the same

D) A little less

E) A lot less

Object-Oriented Programming

- Neat, popular technique
- Makes difficult, complicated things easier and simpler
  - unfortunately, also makes simple things a little harder
- Can help reduce errors if done right

What is an ‘object’?

- A data type, like a structure
- structures have fields
- objects have fields and “methods”
  - objects “perform actions”

OOP

- The general goal is to limit the amount of duplicated code
  - but OOP provides one specific strategy for this

Catalog Example

- We’ve worked with `structs` before
  - Make a “Car” object
    - A car has 4 wheels
    - A car can drive
    - A car has some amount of gas
  - Make a Truck with same properties
  - Make a Bicycle

WHY would you use OOP?

- It’s useful for reducing repeated code
  - i.e., instead of copying & pasting code, you get to “re-use” code you wrote elsewhere
- It’s structured in largely the same way we think about the world

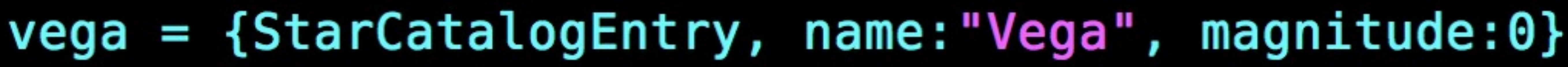
WHEN should you use OOP?

- Any time you can visualize your data in a hierarchy
- Or, any time you’re working with graphics
  - Today’s tutorial will show some nice examples of graphics & objects

Graphics & OOP

- Graphics are nice to think of as object oriented:
  - You have a “plot window” that has data as a “property”
  - You can change that “data property” without re-making the “plot window”

REVIEW: `structures`



```
vega = {StarCatalogEntry, name:"Vega", magnitude:0}
```

Is `vega` an instance or a type?

A) instance

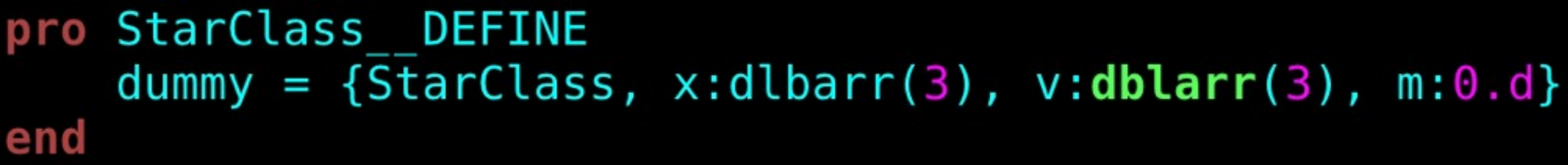
B) type

C) Neither

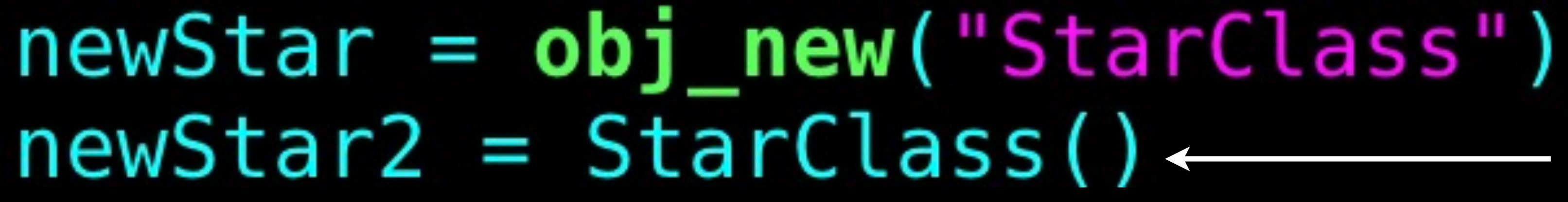
D) I don’t know

Some Syntax

- A lot of the syntax related to OOP is the same as that for structures



```
pro StarClass__DEFINE
    dummy = {StarClass, x:dlbarr(3), v:dblarr(3), m:0.d}
end
```

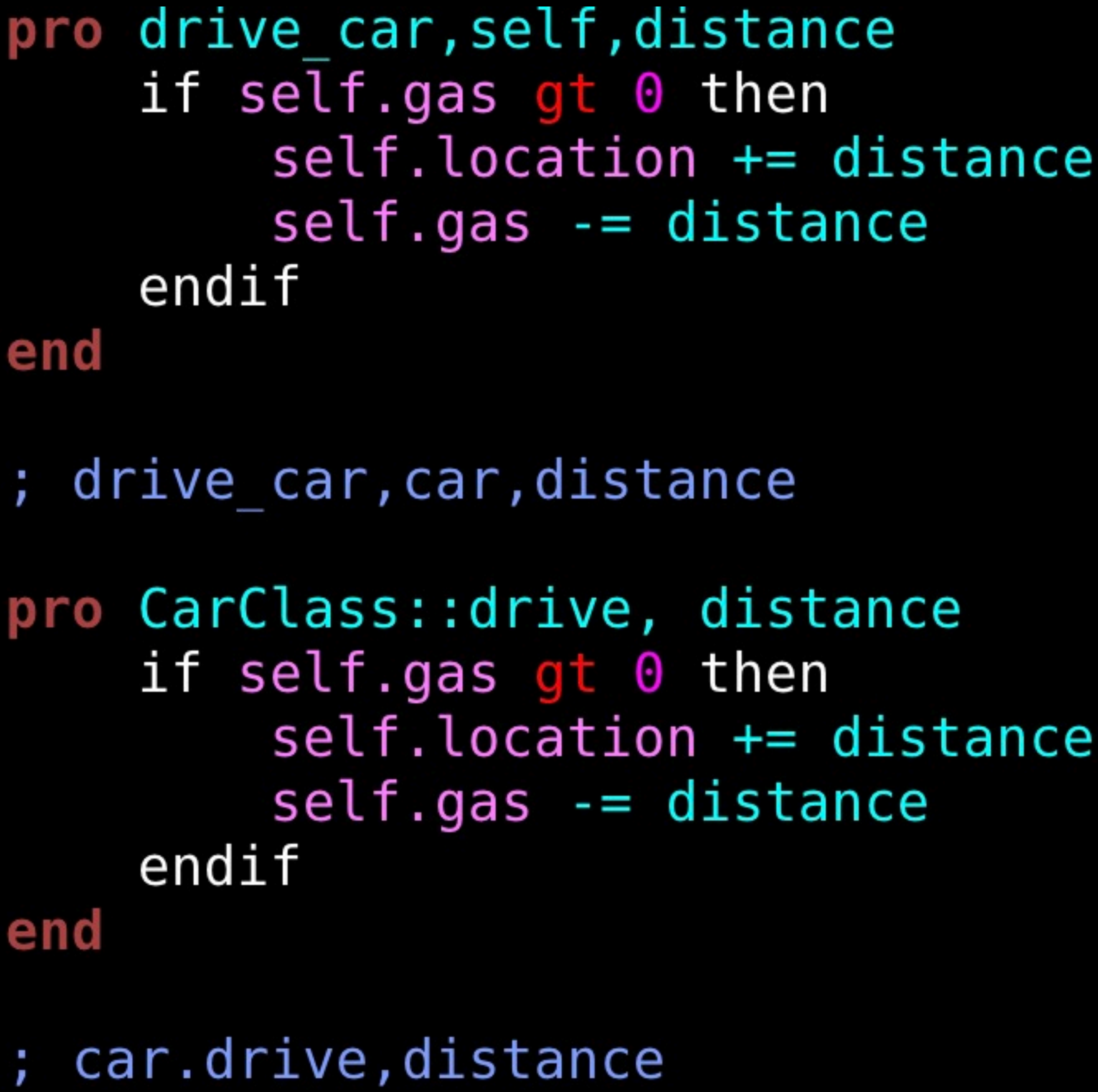


```
newStar = obj_new("StarClass")
newStar2 = StarClass()
```

← New in IDL 8

Objects perform Actions

- Our “Car” can “Drive”
- So we want to define an “action” for the car that is “drive”
  - “Actions” that objects perform are called “methods”
  - Methods are procedures or functions, BUT they have some special behavior



procedure

```
pro drive_car,self,distance
    if self.gas gt 0 then
        self.location += distance
        self.gas -= distance
    endif
end

; drive_car,car,distance
```

method

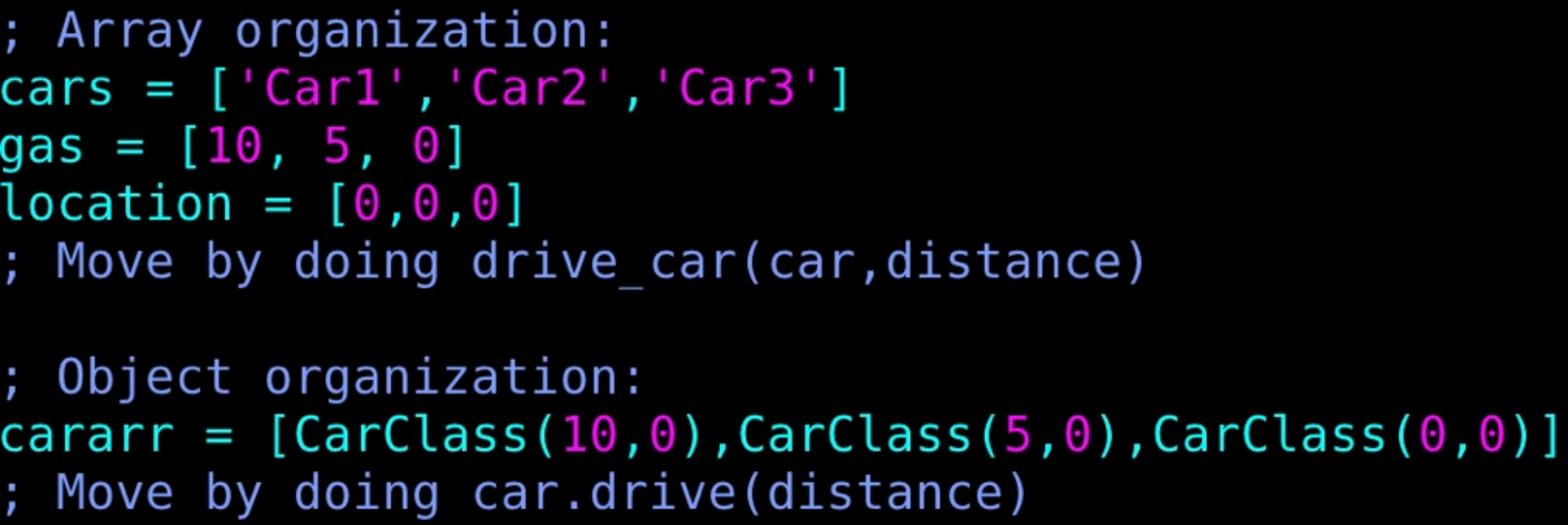
```
pro CarClass::drive, distance
    if self.gas gt 0 then
        self.location += distance
        self.gas -= distance
    endif
end

; car.drive,distance
```

OOP vs “Procedural”

- Both can accomplish the same thing
- OOP is more about *organization*: it’s a way to think about data and code flow
- In OOP, the “state” is stored as part of an “object”
    - The *car* knows how much gas it has, instead of you keeping track of Car #1 and Gas #1

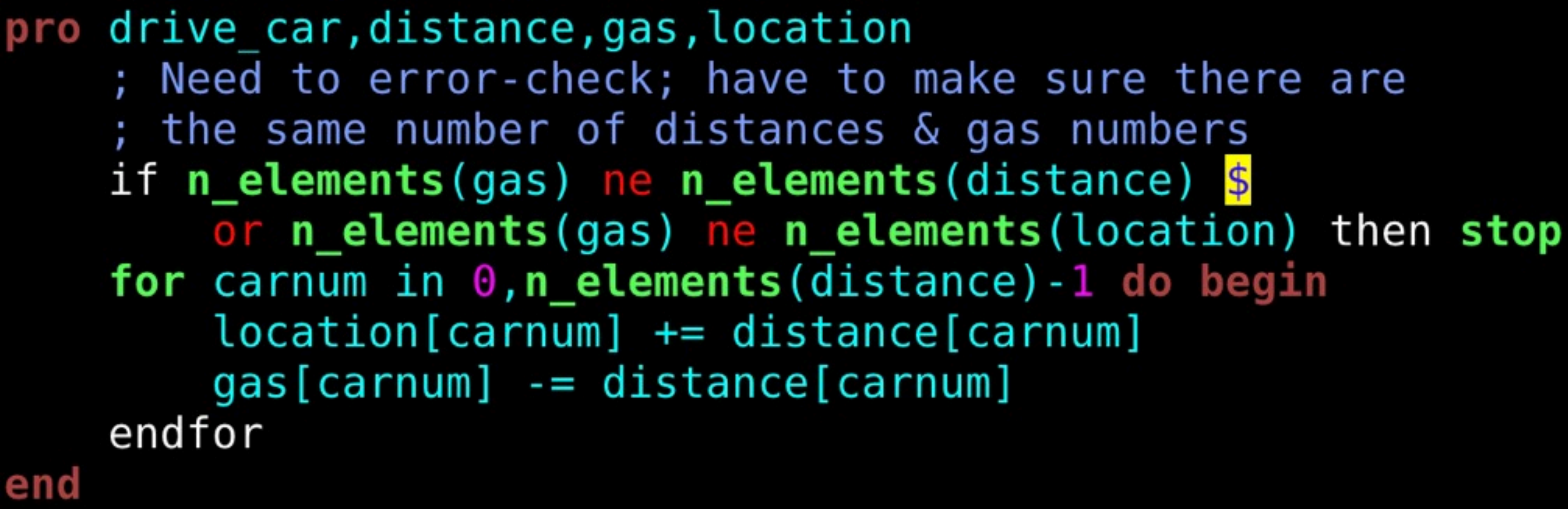
“array” vs “object” thinking



```
; Array organization:
cars = ['Car1','Car2','Car3']
gas = [10, 5, 0]
location = [0,0,0]
; Move by doing drive_car(car,distance)

; Object organization:
cararr = [CarClass(10,0),CarClass(5,0),CarClass(0,0)]
; Move by doing car.drive(distance)
```

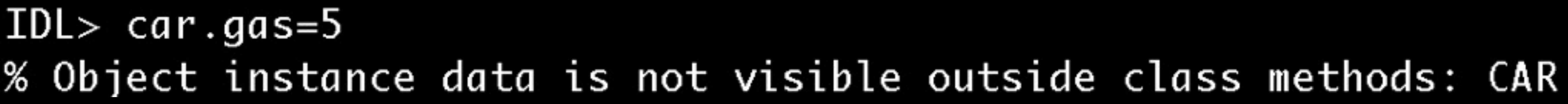
Array style drive procedure....



```
pro drive_car,distance,gas,location
    ; Need to error-check; have to make sure there are
    ; the same number of distances & gas numbers
    if n_elements(gas) ne n_elements(distance) $
        or n_elements(gas) ne n_elements(location) then stop
    for carnum in 0,n_elements(distance)-1 do begin
        location[carnum] += distance[carnum]
        gas[carnum] -= distance[carnum]
    endfor
end
```

Public & Private

- In some languages, including IDL, you cannot change any of the variables in an object



```
IDL> car.gas=5
% Object instance data is not visible outside class methods: CAR
```

- Objects are different from structs!
- This may seem silly, but the assumption is that only the object knows how to change its properties

Methods

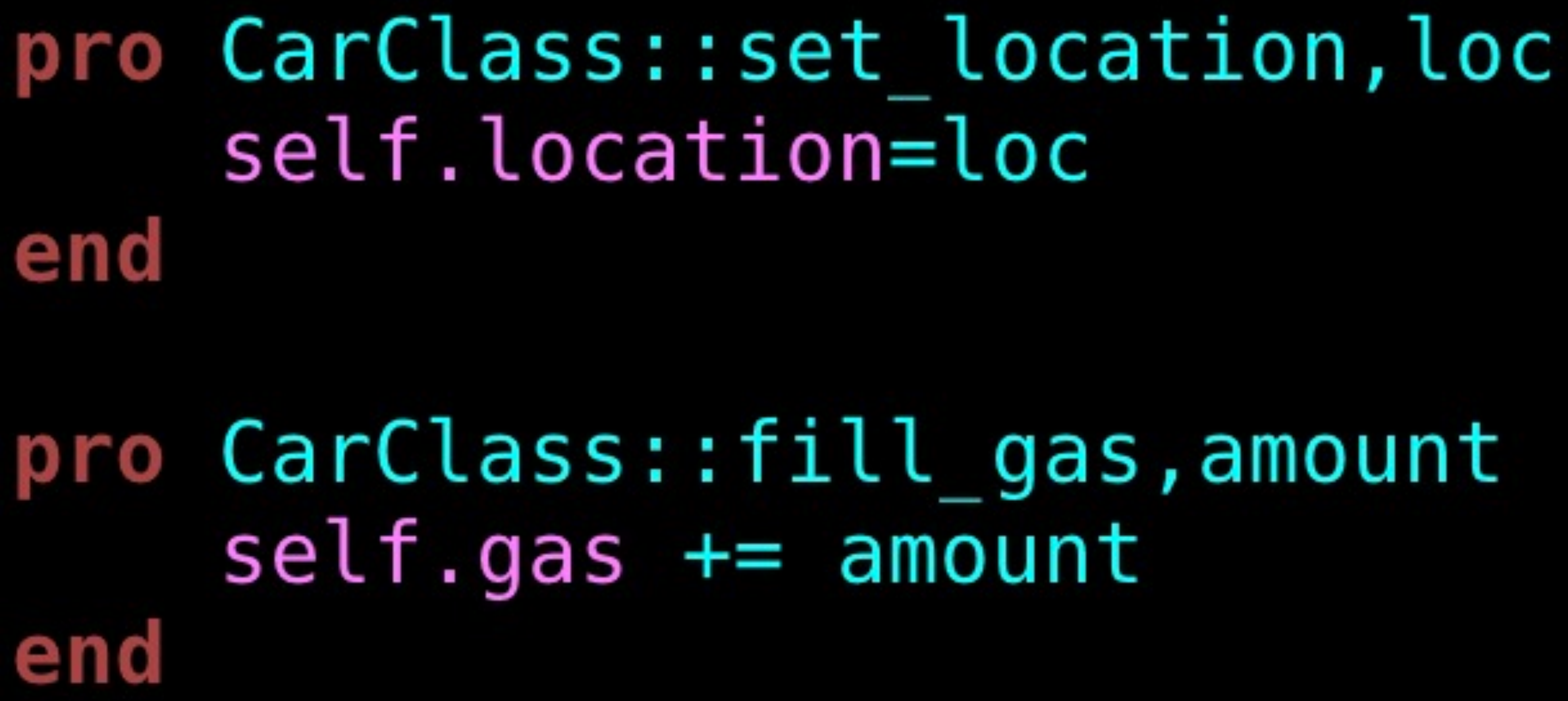
- So instead of `car.gas=5`, you would do `car.set_gas,5` (but you have to define a `set_gas` method)
  - More sensible would be, instead of `car.gas+=5`, do `car.fill_gas(5)`

Public vs Private

- “Public” variables are object fields that anything / anyone can change
  - IDL does not have these (but C and python do)
- “Private” variables are object fields that ONLY object methods can change
  - IDL and C have these, python doesn’t directly

So how do you set values?

- Define methods to do so
- The standard is to use `set_[fieldname]`:



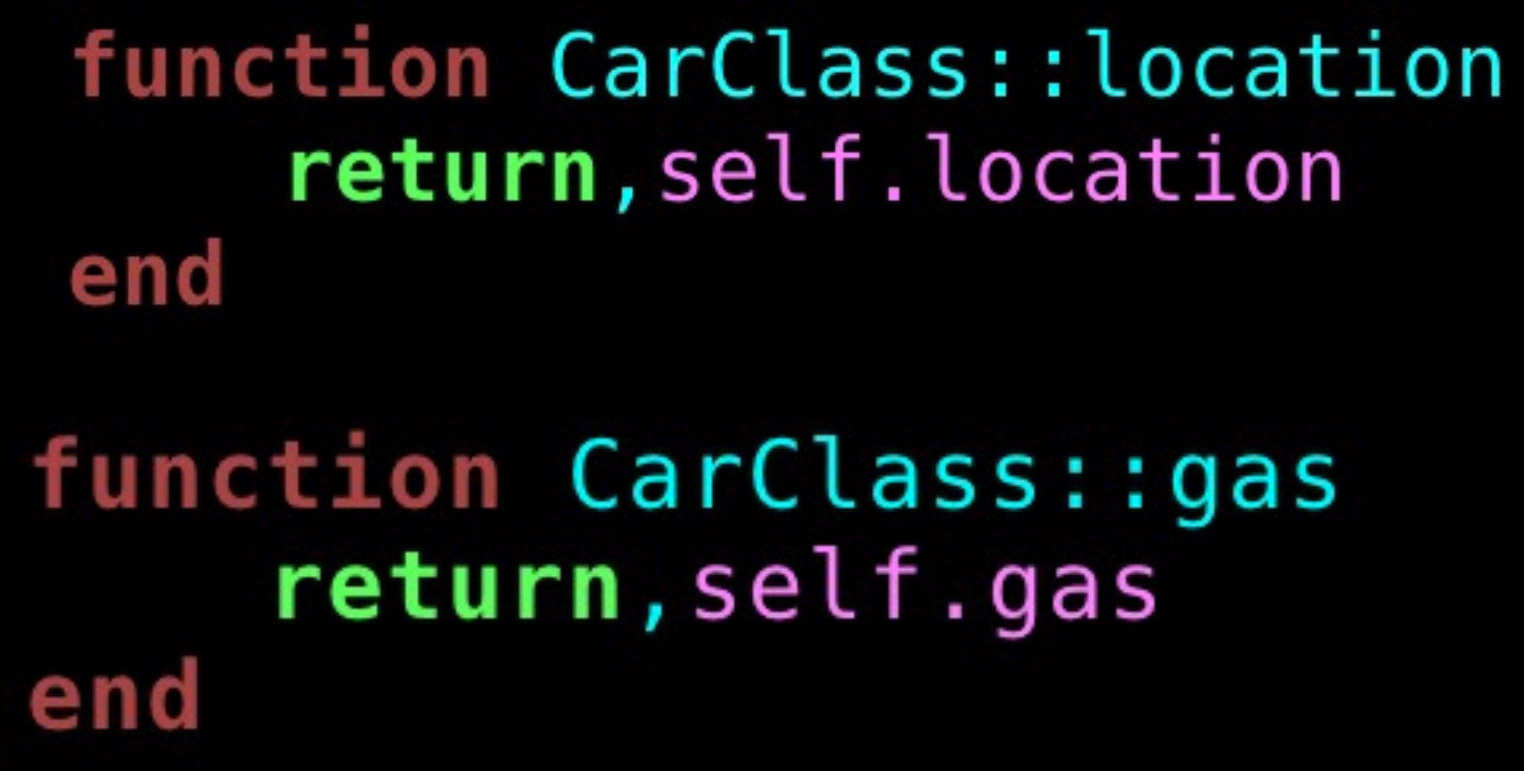
```
pro CarClass::set_location,loc
    self.location=loc
end

pro CarClass::fill_gas,amount
    self.gas += amount
end
```

You also have to get them

- If you want to see a variable, you can’t do “`print,car.gas`”
  - You need to define methods and call them
  - “`print,car.how_much_gas()`”

getters



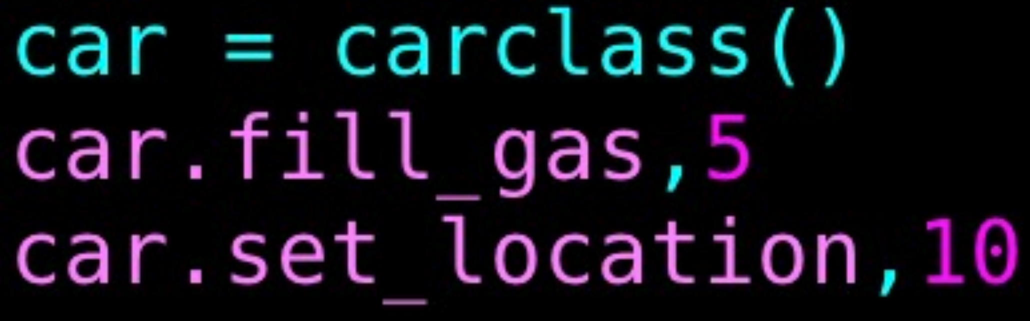
```
function CarClass::location
    return,self.location
end

function CarClass::gas
    return,self.gas
end
```

- `print,car.gas(),car.location()`

Initialization

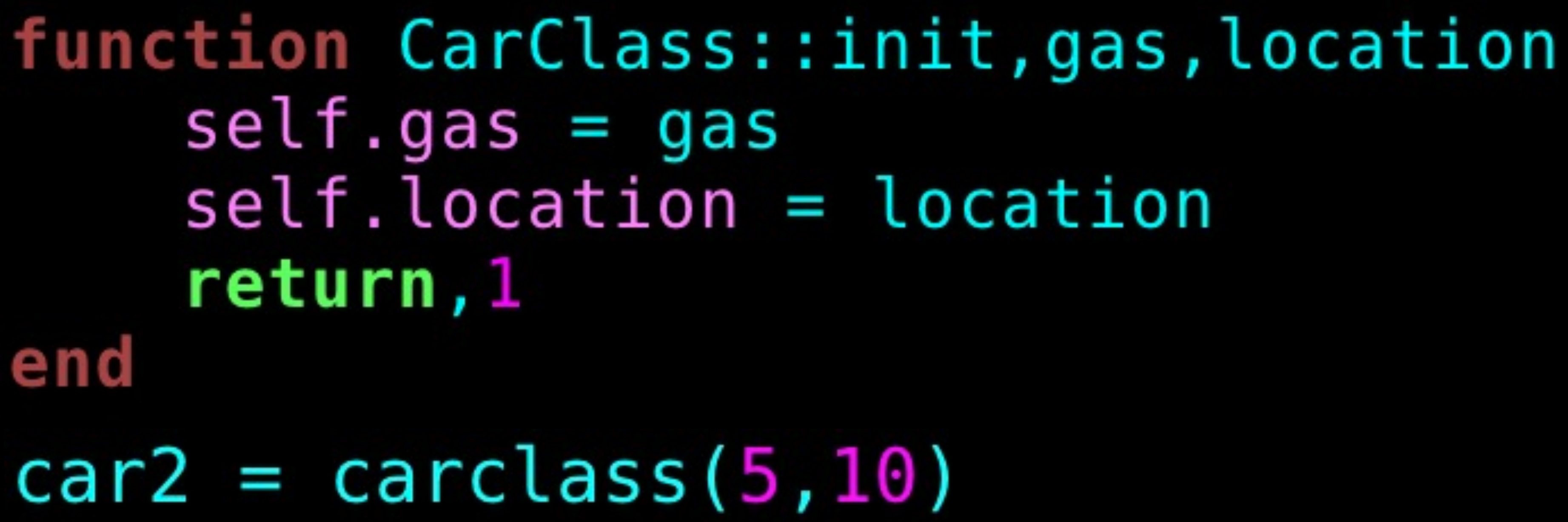
- With all that get & set machinery, it’s kind of a pain to create a new instance of a class:



```
car = carclass()
car.fill_gas,5
car.set_location,10
```

Initialization

- Of course, there’s an easier way: You define an “`init`” method.



```
function CarClass::init,gas,location
    self.gas = gas
    self.location = location
    return,1
end
car2 = carclass(5,10)
```

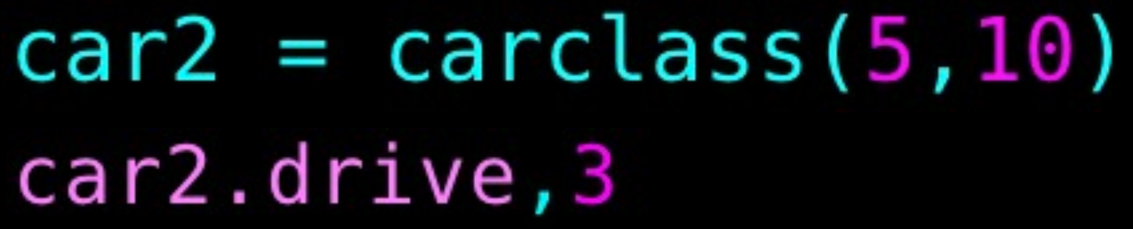
`::init` *must* be a function

If you used exactly this function, you could no longer do

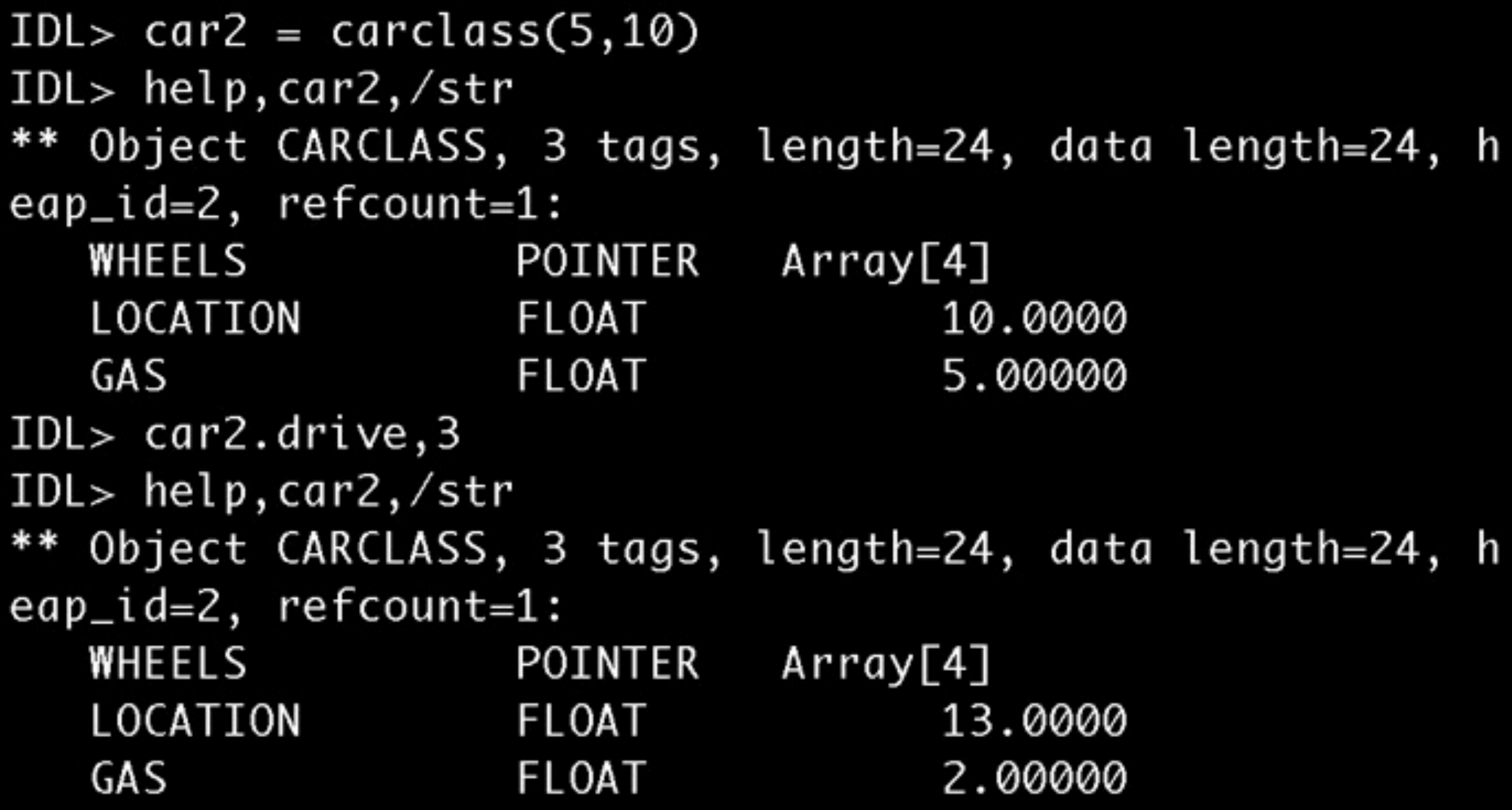
`car = carclass()`

So you have a car...

- Probably want to drive it



```
car2 = carclass(5,10)
car2.drive,3
```



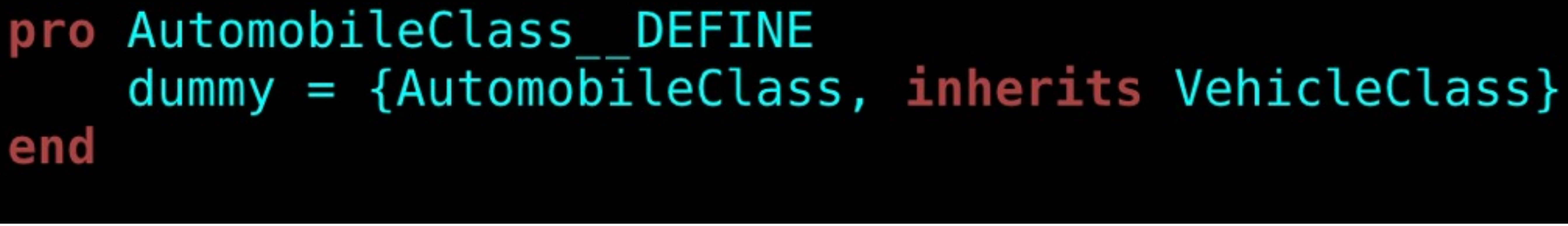
```
IDL> car2 = carclass(5,10)
IDL> help,car2,/str
** Object CARCLASS, 3 tags, length=24, data length=24, h
eap_id=2, refcount=1:
   WHEELS          POINTER   Array[4]
   LOCATION        FLOAT           10.0000
   GAS             FLOAT           5.00000
IDL> car2.drive,3
IDL> help,car2,/str
** Object CARCLASS, 3 tags, length=24, data length=24, h
eap_id=2, refcount=1:
   WHEELS          POINTER   Array[4]
   LOCATION        FLOAT           13.0000
   GAS             FLOAT           2.00000
```

The really cool stuff

- Can also get super confusing
- Inheritance is awesome - it saves you writing a LOT.
- But it also makes it hard to track down errors sometimes
  - still worth it.

Back to hierarchy...

- Say we had made a "`vehicleclass`"
- It has a location, and perhaps a "move" method, but no gas or wheels
  - Then we make an "AutomobileClass" that inherits from "VehicleClass". That would look like



```
pro AutomobileClass__DEFINE
    dummy = {AutomobileClass, inherits VehicleClass}
end
```

Inheritance

- `AutomobileClass` inherits from `VehicleClass`
- `CarClass` inherits from `AutomobileClass`
- A Car is an Automobile and a Vehicle

Inheritance example

- Vehicle
  - Has passengers
- Automobile
  - Like a vehicle, has passengers
  - Has an engine, wheels
- Car
  - Does everything an automobile does
  - Also has a passenger seat