RECAP

- multiple inheritance refers to the ability to inherit from more than one class (or type)

- both conceptually and syntactically multiple inheritance seems to follow naturally from single inheritance

RECAP

- just as with single inheritance, the subclass has access to its parents' behaviour

- behaviour defined lower in the inheritance tree takes precedence (or shadows) anything higher up

`Parent __init__()`

RECAP

- with multiple inheritance, parent __init__s are not automatically called
- rather the relevant ones (if any) need to be specifically called with the applicable arguments
- when calling a parent init from a subclass, self (the instance of the subclass) needs to be passed in as the first argument

Revisiting super()

RECAP

- in multiple inheritance, we're best off relying on super rather than rigidly referencing classes by name
- super returns a proxy (or delegator) object that indirectly references the next parent or sibling in the inheritance chain
- the search order followed, which super() helps traverse, is officially called the method resolution order

RECAP

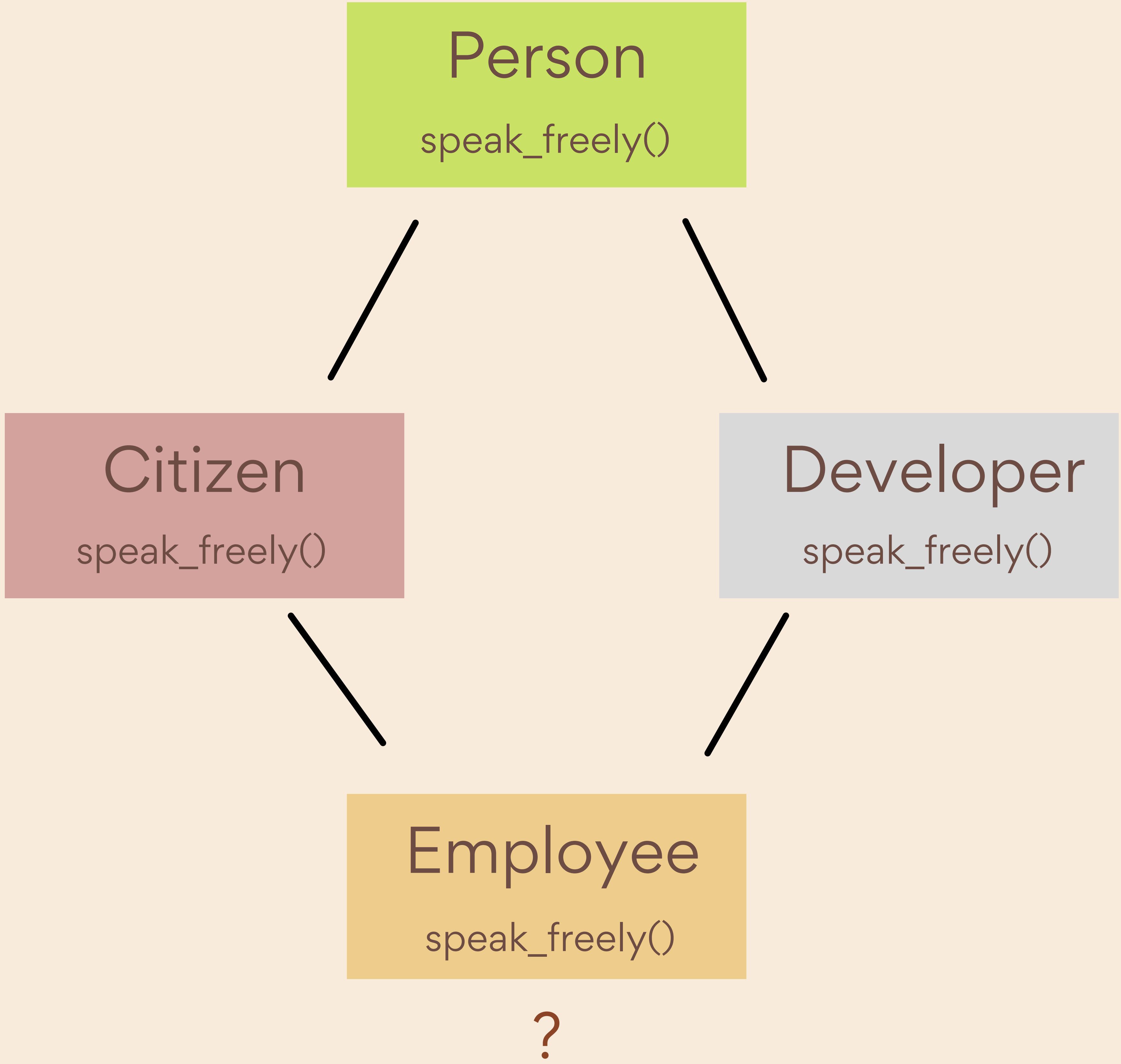
- variadic method signatures allow us to accept an ex-ante unknown number of arguments in our methods

- they are very useful when combined with super() in building cooperative multiple inheritance class hierarchies

The Diamond Problem

ISSUE

- If the **Person** type defines a given method, and
- **Citizen** and **Developer** are inheriting sibling subclasses that both override that method, then
- which implementation does Employee, which inherits from both Citizen and Developer, inherit?



RECAP

- the diamond problem arises in class hierarchies involving multiple inheritance

- specifically, when a subclass inherits from two sibling classes which both define a same-named method or attribute

- python's solution to the diamond problem is to pick the ordinally first sibling from the __mro__

RECAP

- __mro__ is central to working with inheritance in python, especially multiple inheritance
- __mro__ reflects a linear sequence of types to be referenced in that order
- it is the result of a linearization algorithm (C3) meant to produce deterministic results
- the __mro__ reflects 2 rules:
  - children must come before parents
  - siblings need to be searched per the order defined in the subclass

RECAP

- understanding multiple inheritance will make you a more well-rounded python developer, though it shouldn't be our go-to solution everywhere possible

- two excellent use cases:
  - using mixins to shorten inheritance hierarchies, or
  - ABCs to define more complex relationships between types

RECAP

- mixins are superclasses that implement some specific behaviour
- they are not meant to be instantiated independently; rather, only combined with a cooperative base class or other mixins
- to clearly communicate intent, it is a good practice to append "Mixin" to all mixin class names

RECAP

- multiple inheritance is also immensely useful in defining and managing a hierarchy of types

- this is typically done by subclassing ABCs into more specific abstract types using multiple inheritance

- here, multiple inheritance is being used as a code organization tool in order to manage types and their relationships

Skill Challenge #16

#multipleinheritance

Requirements

> Define a new type called CreditCard that supports a generate() method that randomly chooses the middle 14 digits of a credit card. This type should also support a number property that reflects the generated number is spaced blocks of 4 digits, similar to how they would show on a credit card, e.g. 5241 0213 8828 6423

> Then, define two other types that are not meant to be instantiated but rather only cooperatively work with the generate() method from CreditCard:

- VisaMixin, which prepends the digits '42', and
- MasterCardMixin, which prepends the digits '53'

> Then define another type ValidMixin similar to the above that will be responsible for putting in the right checksum. To calculate the checksum, we could use Luhn's algorithm:

- keep track of a cumulative sum, initially 0
- starting from the rightmost digit (i.e. 15th), work leftward
- find the double (2 * n) of every other number, starting with the one above
- if the double is greater than 9, add the sum of its digits to the cumulative sum (e.g. for 12 -> add 1+2 -> 3)
- if not, add the double (2 * n)
- for other non-doubled numbers, add the number itself
- in the end, return the difference between 10 and cumulative sum mod 10

> The integer returned by the above becomes the 16th digit of the credit card if and only if the ValidMixin for valid credit cards is included in a class definition.