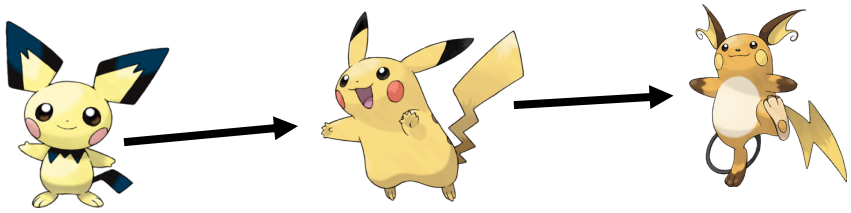
CS 106AX Programming Methodologies in JavaScript and Python Autumn 2026

IG Problem 3A: Reassemble Pokémon Evolutions!

Written by Ben Yan

In this problem, you'll be asked to perform another miracle of reassembly, but this time for discovering the full evolutionary sequences of Pokémon species* ☺. Back to biology class we go! Like all living organisms, Pokémon undergo gradual evolution from one form or species to another. For instance, the baby mouse ***Pichu***, when they become sufficiently happy, evolves into the well-known kawaii ***Pikachu***, who then evolves into the powerful, fully evolved ***Raichu***. In summary, Pichu's evolutionary sequence is

```
Pichu -> Pikachu -> Raichu
```



Other evolutionary sequences, ranging from the water turtle ***Squirtle*** to fire dragon ***Charmander***, include

```
Squirtle -> Wartortle -> Blastoise
Charmander -> Chameleon -> Charizard
Vulpix -> Ninetales
Lugia
```

where ***Lucia*** is a rare species that does not evolve into, or evolve from any other species. And for now (sorry devoted Pokémon fans!), we'll pretend that evolution is entirely linear, and that **each Pokémon species evolves into <u>at most one</u> other species**, without any branching possibilities. Sadness ☹.

Your task here is, given a file where each line details a Pokémon species and who they directly evolve into, parse that file's contents to reconstruct every Pokémon's evolutionary sequence from start to end.

**If you've never heard of Pokémon, in this day and age, Ben would like a conversation with you.*

Part I: File Reading and Creation of a Pokedex Dictionary

Implement a function **`createPokedex`**, which accepts as an input a file structured like the one below.

Each line in the file is formatted **<u>one of two possible ways</u>**:

- **`species:nextSpecies`**, with a colon separating the two components. Here, **`species` directly evolves into `nextSpecies`**, e.g., on Line 1, ***Squirtle*** evolves to ***Wartortle***, and on Line 3, ***Wartortle*** evolves to ***Blastoise***.
- **`species`**, with no colon. This is the name of a species that does not evolve to any other species, e.g., ***Blastoise***.

```
Squirtle:Wartotle
Vulpix:Ninetales
Wartotle:Blastoise
Blastoise
Pichu:Pikachu
Ninetales
Lugia
Raichu
Pikachu:Raichu
```

Note that the lines can be ordered in any way, but you can guarantee the first species in each line is unique. **This function should return a dictionary mapping each Pokémon species to the species they evolve into, or `None` if that species doesn't evolve**. You are not required to perform any error-checking.

For instance, given the exact file above, the returned value would be the dictionary:

```
{
      "Squirtle": "Wartortle",
      "Vulpix": "Ninetales",
      "Wartortle": "Blastoise",
      "Blastoise": None,
      "Pichu": "Pikachu",
      "Ninetales": None,
      "Lugia": None,
      "Raichu": None,
      "Pikachu": "Raichu",
}
```

Use the space below to present your implementation.

```
def createPokedex(filename):
      """
      Given a file structured like the one above, returns a dictionary with
      each Pokemon species as keys, and the corresponding value as the species
      they evolve into, or None if the Pokemon doesn't evolve, e.g., Raichu.
      """
```

Part II: Reassembling Evolutionary Sequences

Next, you'll write a function **`reassembleEvolutions`**, whose input is a **`pokedex`** dictionary like the one returned by **`createPokedex`**. **This new function should return a list of evolutionary sequences**, where each sequence is a list of Pokémon species such that each species evolves to the one right after it.

The last Pokémon in each sequence must be a fully evolved species. That is, it does not evolve **to** any other species. Meanwhile, the first Pokémon in each sequence must not evolve **from** any other species.

As an example, consider the dictionary presented on the prior page and below. Recall that each key is a Pokémon, and its corresponding value is who they directly evolve into, or **`None`** if they've fully evolved.

```
pokedex = {
      "Squirtle": "Wartortle", "Vulpix": "Ninetales", "Pikachu": "Raichu",
      "Wartortle": "Blastoise", "Ninetales": None, "Pichu": "Pikachu",
      "Blastoise": None, "Lugia": None, "Raichu": None,
}
```

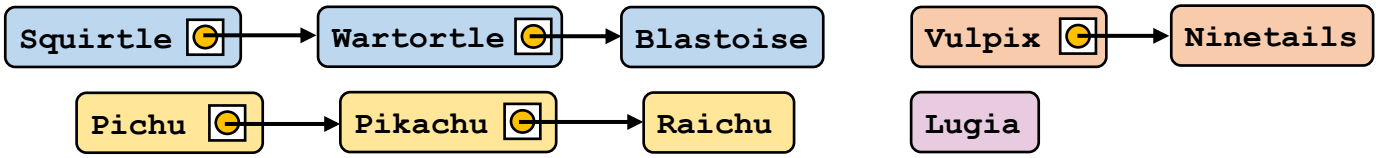
With this input, **`reassembleEvolutions`** may return the following list.

```
[["Squirtle", "Wartortle", "Blastoise"],
 ["Vulpix", "Ninetales"],
 ["Pichu", "Pikachu", "Raichu"],
 ["Lugia"]]
```

To interpret the first inner list, ***Squirtle*** evolves to ***Wartortle***, who evolves to ***Blastoise***. ***Blastoise*** is fully evolved, and ***Squirtle*** does not evolve from any species, meaning we have a **full evolutionary sequence**.

The order of sequences in the returned outer list does not matter. However, along with the requirements stated above, all Pokémon featured in the input **`pokedex`** should be included in exactly one sequence, and your approach should be able to assemble evolutionary sequences of any positive length as needed.

Use the space on the next page to present your implementation.



Suggested Implementation: Starting from each Pokémon key in the **`pokedex`**, trace the sequence of its evolutionary descendants until it becomes fully evolved. For instance, from **`"Pikachu"`**, we'd have **`["Pikachu", "Raichu"]`**; from **`"Pichu"`**, we'd have the overlapping **`["Pichu", "Pikachu", "Raichu"]`**; and from **`"Wartortle"`**, we'd have the sequence **`["Wartortle", "Blastoise"]`**.

For the sequences that overlap, they are part of the same evolutionary line. And **the one thing they are guaranteed to have in common is their final, fully evolved Pokémon** ☺. Note the longer sequence is the more complete one. Thus, as you iterate over the **`pokedex`**, build up a new dictionary, where the **keys are each fully evolved Pokémon**, e.g., **`"Blastoise"`**, **`"Ninetales"`**, **`"Raichu"`**, **`"Lugia"`**, and the corresponding value is the longest evolutionary sequence leading up to them we've observed so far.

```
def reassembleEvolutions(pokedex):
    """
    Given a dictionary mapping Pokémon to the Pokémon they directly evolve
    into, return a list of all distinct evolution sequences, start to end.
    """
```

🫶 IG Problem 3B: Ship Names Generator 🫶

Written by Ben Yan

In fandom, a *ship* is the pairing of two individuals in a story—or occasionally, real life—that fans would love to see in a romantic relationship. Maybe they're already in one, who knows ☺. For instance, Korra and Sato in *Avatar: The Legend of Korra*, Dean and Castiel in *Supernatural,* and Kylo and Rey in the *Star Wars* films.

Many ships are celebrated with a *ship name*, a combined title that's typically formed by some linguistic marriage of their names or clever wordplay. For this problem, we'll implement a quirky Python algorithm:

```
def shipThem(nameOne, nameTwo):
```

which creates a ship name between two first names, **`nameOne`** and **`nameTwo`**, as follows.

- If **`nameOne`** begins with a consonant, we take everything before the first vowel in **`nameOne`**, i.e. the starting stream of consonants, and concatenate it with everything including and after the first vowel in **`nameTwo`**. For instance, **`shipThem("draco", "harry")`** would result in **`"drarry"`.**

  In the event that **`nameTwo`** starts with a vowel, e.g., **`"amity"`**, we include all of it in the ship name, e.g., **`shipThem("luz", "amity")`** becomes the delightful **`"lamity"`**.

- If **`nameOne`** begins with a vowel, we take everything before the **second** vowel in **`nameOne`**, and concatenate it with everything after (**but not including**) the first vowel in **`nameTwo`**. For instance, **`shipThem("elio", "oliver")`**, the leads from *Call Me by Your Name*, becomes **`"elliver"`.**

You may assume every input is lowercase, a single word, and has at least one consonant and vowel. If the name starts with a vowel, you can assume there will be a second one. **We consider *a, e, i, o, u* to be vowels, but not *y*—**so sorry linguists! Here are some additional fan and real ships if you're into that ☺.

Python Interpreter

```
>>> shipThem("scott", "jerry")
"scerry"
>>> shipThem("megumi", "yuji")
"muji"
>>> shipThem("wednesday", "enid")
"wenid"
>>> shipThem("armin", "eren")
"armren"
```



Use the space on the next page to present your implementation of **`shipThem`**, and any helper functions. My approach makes use of at least two, one of which is provided to you, **`isVowel`**, and the other which you're encouraged to write yourself, **`findFirstVowel(word, start)`**. This function can return the index of the first vowel in **`word`** at or after the **`start`** index. To search the entire word, omit the **`start`** parameter.

```
""" This function takes two one-word names, and smashes them together into
    a fanfic-ready ship name, according to the vowel / consonant rules above.
"""
def shipThem(nameOne, nameTwo): # implement this!
```

```
""" Returns index of first vowel at or after the start position, and -1 if
    no vowel could be found. If the start is omitted, it's 0 by default.
"""
def findFirstVowel(word, start = 0): # recommended: implement this!
```

```
""" Returns true if the string is of length one, and a lowercase vowel
"""
def isEnglishVowel(ch):
      return len(ch) == 1 and "aeiou".find(ch) >= 0
```