RECAP

- python is a dynamically typed language
- type checking is only performed when the code is executed, and
- types are not set it stone and could change over the variable's lifetime

RECAP

- duck typing is the application of abductive reasoning to type checking: if it quacks like a duck, it's probably one
- duck typing goes hand in hand with the EAFP style of code (quite popular in the python community)
- this stands in contrast to LBYL style popular in languages like java

RECAP

- in a python OOP context, dynamic protocols refer to a set of special methods that must be supported for a class/type to behave a certain way

RECAP

- pythonic sequences are characterized by their support of 3 types of behaviour:

  - membership testing with the "in" keyword
  - 0-based indexing with square brackets
  - the ability to slice

RECAP

- we could make user-defined classes support the sequence protocol by implementing __getitem__ and __len__

- the protocol, in collaboration with the python interpreter, endows our class with functionality that we haven't explictly implemented, e.g. membership testing with "in" keyword is added even though we did not define a __contains__ separately

RECAP

- slice objects are purpose-built constructs that could be used for pythonic extended indexing

- to fine-tune slicing behaviour in user-defined sequence types, we simply need to refine the __getitem__ implementation for when such objects are provided as input

RECAP

- in programming, iteration refers to applying a set of instructions repeatedly until a condition is met
- everything we could loop over with a for loop in python is an iterable
- an iterator is a special object that does just one thing: returns the next item in an iterable

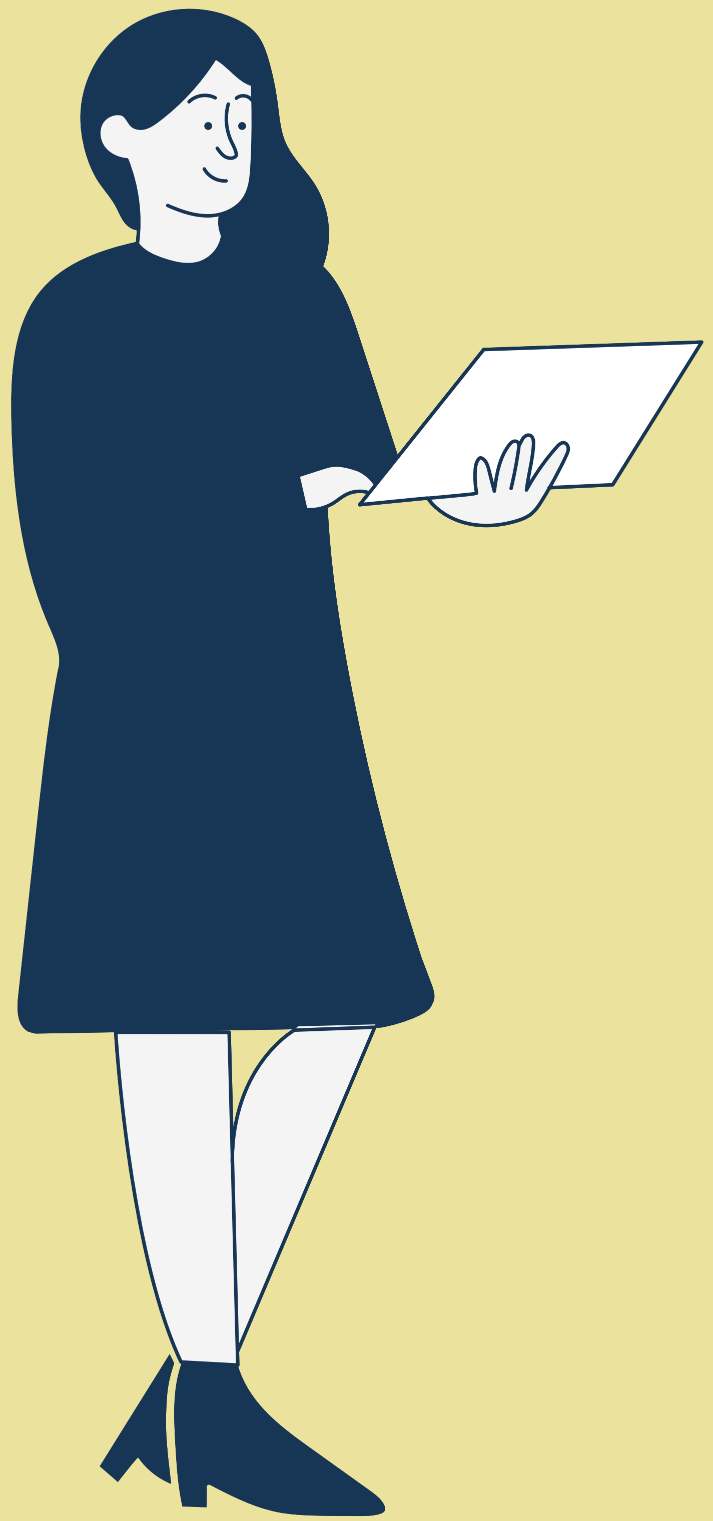
RECAP

- the iterator protocol in python instructs the interpreter on what objects should be considered iterable

- and it really comes down to anything that implements __iter__ and __next__

Skill Challenge #14

#dynamicprotocols



andybek.com

Requirements

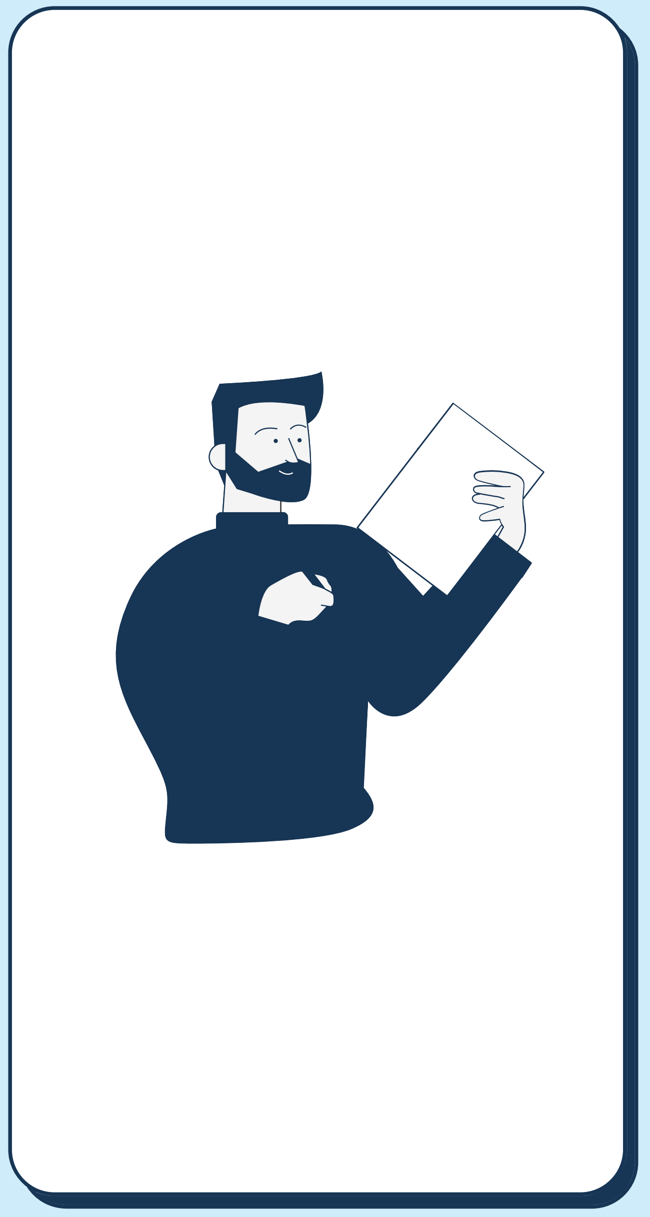
> Define two new types: PlayingCard and CardDeck

> PlayingCard instances should support two instance attributes: suit and rank. The specified attributes should be standardized to lowercase, but not otherwise validated. Equality and a good string representation should be supported.

> CardDeck will represent the traditional deck of 52 cards:
  - 4 suits ('spades', 'diamonds', 'clubs', 'hearts') *x*
  - 13 ranks (2 through 10, inclusive, Ace, Jack, King, Queen)

> CardDeck should be a pythonic sequence supporting the following functionality:
  - membership testing with 'in' keyword (returns bool)
  - square bracket indexing with ints (PlayingCard instances should be returned)
  - slicing with slice objects (CardDeck should be returned)
  - iteration in for loops
  - concatenation through '+' operator, supported between two instances of CardDeck, or PlayingCard & CardDeck
  - concatenation/extension through '*' operator, supported between an instance of CardDeck and int
  - both of the above concatenation operations should be commutative

Requirements Continued



> If instantiated with no args, CardDeck should generate a full deck of 52 cards. If instantiated with a python list of at least one PlayingCard, CardDeck should reflect only the list of cards passed to it at instantiation. If a python list containing no valid PlayingCard instances, the default full deck of 52 cards should be generated.

> Finally, CardDeck should support a draw(n=1) method that randomly selects a card from a given deck and returns it without replacement, i.e. the number of cards remaining in the deck decreases by 1

> If a user draws more than 1 card at a time, e.g. draw(n=3), a CardDeck type should be returned; if only one card is drawn, a PlayingCard type should be returned.