recap

1. inheritance offers us a mechanism for creating new classes that modify or extend the behavior of existing classes
2. inheritance establishes hierarchies between classes, e.g. super/base/parent class -> sub/child class
3. well-designed inheritance hierarchies define "is a" relationships between classes

recap

1. one of the most important uses of inheritance is to add or modify functionality from an existing class without modifying that class
2. this contributes to code reuse and organization as well as object hierarchies that are easy to reason about

recap

1. all python classes implicitly inherit from object, whether that is specified in the class definition or not

2. this inheritance guarantees certain base behavior in all subclasses, like the fact they're callable, have some default representation, and more

3. object really exists to provide these reasonable, barebones defaults to all subclasses, so as to enable child classes to only customize or extend only what is relevant to the use case, rather than re-implement everything

recap

1. attribute and method lookup follows a well defined order:
   Instance -> Class -> Superclass -> object

2. the lookup stops on first match, i.e. it's possible for our method/attribute name to show up more than once in the lookup chain, but those later matches are not reachable; first match wins

3. this lookup is the reason we have the ability to reference attributes or call methods defined in classes (or their parents) from within instances of subclasses

4. the lookup could be easily sourced from the read-only __mro__ attribute, available on the class

recap

1. descendant classes are always checked before their ancestors
2. if a descentant implements a method having the same name as its ancestor class, the descendant's implementation will take precedence
3. this applies to plain attributes as well as methods
4. the mro lookup rules could be used flexibly to model various types of interplay between child and parent implementations

recap

1. the pythonic way to reference the parent class from a subclass is to use the super() built-in
2. super() returns an instance of the parent class, enabling us to directly reference its methods and attributes
3. super() could be called within any subclass method as well as at any step, i.e. not necessarily the first thing we do

recap

1. in python, if the subclass defines a __init__, it is its responsibility to properly initialize all applicable attributes

2. if the subclass does not define a __init__, python automatic invocation at instance initialization will result in the parent's init, if applicable, being called

Skill Challenge #7

#inheritance

Requirements

> Define a new type called BankAccount that takes a single instance attribute: initial_balance (defaults to 0)

> This type should support *deposit*() and *withdraw*() methods, which in turn should only support transactions in positive amounts, i.e. an attempt to deposit or withdraw -2$ should be ignored

> Define 3 more specialized types of BankAccount with the following characteristics:

1. Savings: has pay_interest() method which deposits directly into the account when called; interest rate: 0.0035
2. HighInterest: like Savings, but higher interest and with a withdrawal fee. The fee is specified at initialization and defaults to $5. It is taken from the account's balance on every withdrawal. Interest rate: 0.007
3. LockedIn: like HighInterest, but higher interest without the ability to withdraw on demand. Interest rate: 0.009

> The balance of any of the above accounts should be available by attribute access syntax, e.g. account.balance

> The representation of any of the instances should simply indicate the type of account and the amount contained within, e.g. "A SavingsBankAccount with $100 in it."

recap

1. properties defined in the parent could be extended/modified in the subclass
2. because properties live in the namespace of the class in which they're defined, referring to the them from the subclass requires the use of a fully qualified name in the subclass, i.e. one that specifies the parent class name
3. when inheriting properties, we reserve the flexibility to only extend/modify the methods we care about, e.g. any combination of get, set, del

recap

1. in python, we could extend the behavior of built-in data structures just like we do with regular user-defined ones

2. to do that, we simply define a subclass that inherits from the respective built-in, and modify or enhance only the behaviour we care about

recap

1. built-in types could be extended both to add functionality as well customize existing behavior
2. this includes subclass overrides of specific dunders or the definition of new methods and attributes
3. some commonly extended built-ins are: list, set, dict, and str
4. int and float are also often inherited in creating numeric abstractions with custom behavior

recap

1. extending built-ins directly in python could be tricky because the interpreter makes no guarantees that our overrides will take precedence over the high-efficiency parts of the code implemented in low-level C

2. some common side effects of this are inert overrides (they are ignored) and inconsistent interfaces (spotty coverage)

3. to sidestep all of this, python makes available some special wrappers under the collections module that are easily extensible

inheritance is not the most important thing in OOP

AN ALTERNATIVE:

COMPOSITION

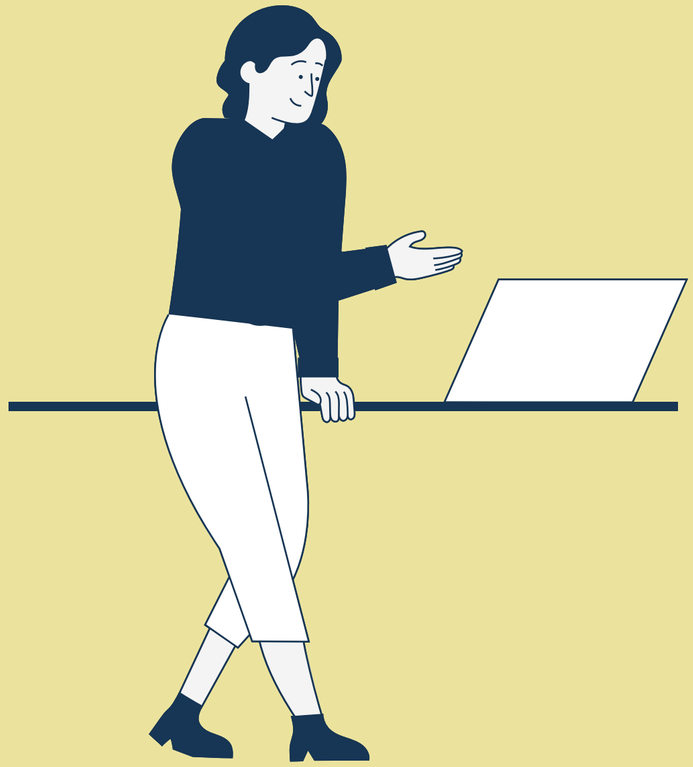
one object is made to contain others and produce more complex constructs.

RECAP

use inheritance for objects that naturally align in "is-a" relationships

prefer composition for those that conceptually fit "has-a" relationships

Skill Challenge #8



#inheritance

Requirements

> Define a new type of dictionary called BidirectionalDict

> It should behave like a regular dictionary, except for enabling the user to look up in either direction, i.e. either a value by key (as in a regular dict) or a key by its value

> The length of the dictionary should reflect the number of unique non-mirrored key-value pairs

> When/if a given pair is removed from the dict, its mirrored sibling should also be removed

> Other dictionary methods like pop() and update() should work as expected, and apply to the mirrored pair too