# recap

1. in python accessor methods—getters and setters—for our attributes are best added when needed

2. it's considered best practice to start with plain, public attributes and evolve to controlled access only as needed

# recap

1. it is often necessary to implement logic around the retrieval, binding and re-binding of attributes, e.g. validation, caching, retrieval counters, etc

2. for such use cases, one Java-like approach is to implement getters and setters for each attribute

# recap

1. remember, attributes and methods defined on a python class are public
2. there are no private or protected attributes per se
3. as programmers we communicate our intent by relying on naming conventions, of which two important ones are:

attributes beginning with a **single leading underscore** are considered **private**

attributes beginning with **two leading underscores** are name **mangled** with the class name

# recap

1. properties are attributes with special behavior
2. they wrap functionality around instance attributes while providing a simple, public interface using dot notation
3. properties help us avoid unnecessary use of getters and setters while keeping the code clean, short and pythonic

# recap

1. all the properties we define live in the class' mappingproxy, not the instance dictionary

2. instance attributes that have the same name as the property do not shadow the workings of the property

   ↘

   in other words, properties take precedence over instance attributes of the same name

4. the mechanics of this are driven by the fact that properties are descriptors, which we'll discuss in depth in a future section

# Skill Challenge #5

#properties

# Requirements

> Define a new type called DNABase which takes a single arg (nucleotide) at initiation

> Internally the value specified in the nucleotide arg should be validated and standardized

> The class should expose the nucleotide in an attribute called base

> The valid bases are: adenine, cytosine, guanine, and thymine

> A user should be able to specify either the full name as above, or the first letter, case insensitively

> The name exposed under the base attribute should be the full name in lowercase regardless of how it was specified

> Invalid or otherwise unrecognized bases should be rejected, whether they are specified at instance creation or later altered through attribute setters

> The new type should have a full representation

# recap

1. a more common syntax for defining properties relies on decorators

2. to use this syntax:

    - the getter is decorated with @property
    - the setter with @property_name.setter
    - all methods carry the name of the property

a **decorator** is a function that takes another function as an argument, adds some functionality, then returns it, and does all of this without otherwise changing the function

# recap

1. parameters to the property constructor are optional: only what we define is supported
2. to support a read-only attribute, we simply define the getter
3. to define a write-only attribute, we only define the setter
4. when a property() parameter has not been set, the corresponding operation raises an AttributeError

# recap

1. python gives us a mechanism through which we could support attributes that look plain but work like functions
2. these are attributes that don't have a variable supporting them but are instead calculated dynamically
3. practically, to associate a function with what looks like a plain attribute, we simply define a read-only property
4. this pattern goes by the following synonymous names: managed properties, computed attributes, computed properties

# recap

1. property-managed attributes are undeletable unless their deleter is explicitly defined
2. the deleter could be added using the @property_name.deleter decorator syntax, or as the third argument fdel to the built-in property constructor
3. property deleters are used to delete the supporting variable from the instance, rather than the property form the class namespace

# recap

1. we add property docstrings by defining them for the getter of the property

2. because docstrings for setter or deleters are ignored, it's good practice to document the property holistically in the getter

# Skill Challenge #6

#properties

# Requirements

> Define a new type Tablet that takes a single attribute called *model* at instantiation

> The class should only support 3 types of models: lite, pro and max

> Each instance will also have *base_storage, added_storage* and *memory* attributes, which will be set automatically depending on the specified model; *base_storage* doubles at each model increment (i.e. 32 of lite, 128 for max), whereas memory increases by 1 (i.e. 2 for lite, 4 for max); *added_storage* is by default 0 for all models

> In addition, the Tablet class should enable users to expand the storage through 2 separate interfaces:

1. an *add_storage*() method, e.g. t1.add_storage(32) should add 32GB to the *added_storage* of the instance
2. direct attribute setter, e.g. t1.storage = 256, should ensure that the overall memory of the device is 256 by dynamically handling the split between *base_storage* and *added_storage* depending on the model

> All models should not exceed 1024GB in combined storage

> *memory* and *base_storage* should be read-only attributes since they can't change

> Tablet instances should have a representation that would make it easy to recreate the instance