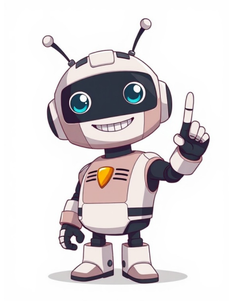
I'm not a robot



Descriptors are objects in a class namespace that manage instance attributes, such as slots, properties, or methods. For example: class HasDescriptors: __slots__ = 'a_slot' # creates a descriptor def a_method(self): # creates a descriptor "a regular method" @staticmethod # creates a descriptor def a_static_method(): "a static method" @classmethod # creates a descriptor def a_class_method(cls): "a class method" @property # creates a descriptor def a_property(self): "a property" # even a regular function: def a_function(some_obj_or_self): # creates a descriptor "create a function suitable for monkey patching" HasDescriptors.a_function = a_function # (but we usually don't do this) Pedantically, descriptors are objects with any of the following special methods, which may be known as "descriptor methods": __get__: non-data descriptor method, for example on a method/function __set__: data descriptor method, for example on a property instance or slot __delete__: data descriptor method, again used by properties or slots These descriptor objects are attributes in other object class namespaces. That is, they live in the __dict__ of the class object. Descriptor objects programmatically manage the results of a dotted lookup (e.g. foo.descriptor) in a normal expression, an assignment, or a deletion. Functions/methods, bound methods, property, classmethod, and staticmethod all use these special methods to control how they are accessed via the dotted lookup. A data descriptor, like property, can allow for lazy evaluation of attributes based on a simpler state of the object, allowing instances to use less memory than if you precomputed each possible attribute. Another data descriptor, a member_descriptor created by __slots__, allows memory savings (and faster lookups) by having the class store data in a mutable tuple-like datastructure instead of the more flexible but space-consuming __dict__. Non-data descriptors, instance and class methods, get their implicit first arguments (usually named self and cls, respectively) from their non-data descriptor method, __get__ - and this is how static methods know not to have an implicit first argument. Most users of Python need to learn only the high-level usage of descriptors, and have no need to learn or understand the implementation of descriptors further. But understanding how descriptors work can give one greater confidence in one's mastery of Python. In Depth: What Are Descriptors? A descriptor is an object with any of the following methods (__get__, __set__, or __delete__), intended to be used via dotted-lookup as if it were a typical attribute of an instance. For an owner-object, obj_instance, with a descriptor object: obj_instance.descriptor invokes descriptor.__get__(self, obj_instance, owner_class) returning a value This is how all methods and the get on a property work. obj_instance.descriptor = value invokes descriptor.__set__(self, obj_instance, value) returning None This is how the setter on a property works. del obj_instance.descriptor invokes descriptor.__delete__(self, obj_instance) returning None This is how the deleter on a property works. obj_instance is the instance whose class contains the descriptor object's instance. self is the instance of the descriptor (probably just one for the class of the obj_instance) To define this with code, an object is a descriptor if the set of its attributes intersects with any of the required attributes: def has_descriptor_attrs(obj): return set(['__get__', '__set__', '__delete__']).intersection(dir(obj)) def is_descriptor(obj): """obj can be instance of descriptor or the descriptor class""" return bool(has_descriptor_attrs(obj)) A Data Descriptor has a __set__ and/or __delete__. A Non-Data-Descriptor has neither __set__ nor __delete__. def has_data_descriptor_attrs(obj): return set(['__set__', '__delete__']) & set(dir(obj)) def is_data_descriptor(obj): return bool(has_data_descriptor_attrs(obj)) Builtin Descriptor Object Examples: classmethod staticmethod property functions in general Non-Data Descriptors We can see that classmethod and staticmethod are Non-Data-Descriptors: >>> is_descriptor(classmethod), is_data_descriptor(classmethod) (True, False) >>> is_descriptor(staticmethod), is_data_descriptor(staticmethod) (True, False) Both only have the __get__ method: >>> has_descriptor_attrs(classmethod), has_descriptor_attrs(staticmethod) (set(['__get__']), set(['__get__'])) Note that all functions are also Non-Data-Descriptors: >>> def foo(): pass ... >>> is_descriptor(foo), is_data_descriptor(foo) (True, False) However, property is a Data-Descriptor: >>> is_data_descriptor(property) True >>> has_descriptor_attrs(property) set(['__set__', '__get__', '__delete__']) Dotted Lookup Order These are important distinctions, as they affect the lookup order for a dotted lookup. obj_instance.attribute First the above looks to see if the attribute is a Data-Descriptor on the class of the instance, If not, it looks to see if the attribute is in the obj_instance's __dict__, then it finally falls back to a Non-Data-Descriptor. The consequence of this lookup order is that Non-Data-Descriptors like functions/methods can be overridden by instances. Recap and Next Steps We have learned that descriptors are objects with any of __get__, __set__, or __delete__. These descriptor objects can be used as attributes on other object class definitions. Now we will look at how they are used, using your code as an example.The provided code snippet is an example of using Python's property decorator to manage access to instance attributes. The `@property` and `@celsius.setter` decorators are used to define getter and setter methods for the `celsius` attribute, respectively.A common use case for this approach is when working with objects that need to enforce certain rules or constraints on their data. For instance, the `Temperature` class uses a property decorator to ensure that the `celsius` attribute can only be set to a float value.Here's an example of how you might extend this code to include additional features:```pythonclass Temperature: def __init__(self): self._celsius = 0.0 @property def celsius(self): return type(self)._celsius @celsius.setter def celsius(self, value): type(self)._celsius = float(value)# Create two instances of the Temperature classt1 = Temperature()t2 = Temperature()# Set and print the initial temperature for both instancest1.celsius = 25.0print(t1.celsius) # Output: 25.0t2.celsius = -10.5print(t2.celsius) # Output: (-10.5,)# Attempting to delete the attribute raises an AttributeErrortry: del t2.celsiusexcept AttributeError as e: print(e) # Output: can't delete attribute# Attempting to set a non-numeric value raises a ValueErrortry: t1.celsius = '25'except ValueError as e: print(e) # Output: invalid literal for float(): '25'# Extending the Temperature class with additional featuresclass HistoricalTemperature(Temperature): def __init__(self, temperatures): super().__init__() self._temperatures = temperatures @property def temperatures(self): return self._temperatures @temperatures.setter def temperatures(self, value): self._temperatures = value# Create a new instance of the HistoricalTemperature classhistory = HistoricalTemperature([20.0, 30.0, -5.0])# Set and print the initial temperature for the historical instancehistory.temperatures = [25.0, -10.5, 15.0]print(history_temperatures) # Output: [25.0, (-10.5,), 15.0]###The key to successfully applying a stash in Git is understanding the different operations involved. A stash is essentially a temporary commit that holds changes to be applied later. When you run `git stash apply`, Git attempts to replicate the original commit's changes on your current working tree.However, there are cases where things get tricky. For instance, if your working tree already has staged or unstaged changes, it can affect how the `git stash apply` command works. The `--index` flag helps address this issue by preserving the staged and unstaged changes in your working tree before applying the stash.There's also a risk of diverging branches when you're dealing with complex cases, like multiple stash applications or advanced Git operations. In such situations, it's essential to start in a clean working tree and use techniques like `git reset` to avoid conflicts.Another valuable tool is `git stash branch`, which allows you to create a new branch based on the exact commit where the original stash was created. This ensures that you can successfully apply the stash with `--index`.For those interested in performance optimization, extensions like the one mentioned for SQL Server 2008/2012 provide useful insights into table partitions and their sizes in MB and GB.###To troubleshoot DHCP issues on Windows, use the "arp" command to find IP addresses attached to the local ethernet segment. However, the networking stack only knows about other hosts if it has previously seen traffic from them. You might need to send a broadcast packet to elicit some sort of response and populate the local ARP table.When dealing with large Active Directory groups, use Get-ADUser with the -LDAPFilter parameter instead of Get-ADGroupMember. This can handle over 5000 entries and supports complex filters.For Microsoft Graph integration issues, ensure that you have imported the necessary module (Import-Module Microsoft.Graph.Applications) and connected to the service (Connect-MgGraph). If encountering an error with New-MgUser or Get-MgUser, try rechecking your syntax or path.In Python, use list slicing to retrieve specific elements from a list. For instance, accessing the first element of a list can be achieved using `liste[0:1]`. You can also use default values if the list is empty by combining this with the `or` operator. For example, `value = (liste[0:1] or ('default',))[0]`.When working with environment variables in .NET, use Environment.GetEnvironmentVariable to retrieve and set variable values. Be aware that different types of storage exist for environment variables, including Process, User, and Machine.In Kubernetes, a bash script can export all YAML files by retrieving their names using `kubectl get -o=name` and then getting each resource's YAML using `kubectl get`. The script can write these to separate files or folders as desired.Lastly, Python's collections library offers the Counter class for counting occurrences of elements in an iterable. This can be useful for tasks like letter frequency analysis in a string. For instance, `Counter({' ': 8, 'o': 4, ...})` will return a dictionary-like object with the counts for each letter.###ARTICLEThe worthiness of a specific key, or the fallback value provided as default_value, plays a crucial role in determining its value. Initially, two parameters are introduced: the first being the key itself you're seeking to locate, and the second being the predefined default option that gets applied when this particular key is not found present.

How to get edgenuity answers 2022. How to get all the answers for edgenuity. How do you get the answers to edgenuity.