When you try to add some types of data to a Python set, you get an error message **`unhashable type`**. What does that mean? This page describes the concept of hashing at a high level and explains why you get this error message.

Sets

Python sets exploit the fact that you cannot store duplicate elements and they do not need to preserve order to allow a more efficient internal representation. In particular, sets are stored in such a way that you can test for membership quickly.

With a list, for instance, you would have to search the entire list in order to determine whether or not a particular element is contained in the list. Therefore, the time it takes to test for membership in a list can be proportional to the size of the list. With a set, a structure called a *hash table* is used to make it such that the time it takes to test for membership is relatively constant no matter how many items are in the set.

With small collections, this timing difference may not matter. Sets still provide advantages when you want to automatically eliminate duplicates in that case. But, when the collections get large, the fast membership operations can be a big win.

Hashing

Hashing makes all of this possible. Without getting into too much detail, a hash table is just a table in which you look things up directly by index into that table. In order to use such a table, you need to be able to determine a consistent index for an arbitrary item (such as a string) that you want to store in the table. This is where *hashing* comes in. Python includes a set of hash functions, one for each *hashable* type. These hash functions take a Python object as input and return a *hash*, which can then be used as an index into a hash table. These hash functions have multiple important properties, including:

1. The hash function will always return the same hash for objects that have the same value.
2. The hash function will distribute the hash values across the possible space.

The first property is important. If I want to be able to check whether a particular element, say **`"Python"`** is in a set, I have to know which index to use to access the hash table. I need to always look in the same place, or I will not be able to find it!

The second property is also important, but for efficiency, not for correctness. If every object had the same hash, we would still be able to find objects. But, they would all be in the same place, so we would have to look through a lot of objects to determine if the one we were looking for is there. If two different objects have the same index into the hash table, this is called a *collision*. There are many ways of handling collisions, but they all require additional overhead to access all of the items that have collided. Therefore, a good hash function avoids collisions as much as possible.

Hash tables and hash functions are a complex subject, but this gives you a high level overview of what is happening so that you can better understand Python sets.

Hashability

Given this discussion, it should be clear that mutable objects are unhashable. If the object changes, you will not be able to find it again in the hash table, as its hash value will also change. So, only immutable objects can successfully be stored in a hash table, and therefore a Python set. Python has a notion of *hashable* and *unhashable* types. A type is hashable if there is a hash function that can be used for that type, which generally means the immutable builtin types. If you try to add an object of an unhashable type to a Python set, you will get the **`unhashable type`** error message. And now you know why!

Dictionaries

Python dictionaries also use hash tables internally for the same reasons as Python sets. You cannot have duplicate keys in a dictionary and dictionaries do not preserve order. Therefore, you can use a hash table and get the same performance benefits. You simply need to store values in the hash table along with the keys.

✓ Complete