Theory: Hashable

34 minutes 6 / 7 problems solved

Start practicing

1395 users solved this topic. Latest completion was 27 minutes ago.

§1. What does "hashable" mean?

You may remember that when discussing dictionaries in Python, we mentioned that not every object can be a key in a dictionary. In Python, only **hashable** objects can be dictionary keys (or set members).

The notion of hashable objects is, naturally, connected to the **hash table** concept. According to official Python documentation, an object is **hashable** if it has a **hash value** that doesn't change during its lifetime and can be compared to other objects. In practice, it means that to be hashable objects need methods `__hash__()` and `__eq__()`. Both these methods are needed because the hash table can only guarantee that equal objects have the same hash value. Unequal objects may also have the same value, so we need additional constraints.

Why only hashable objects can be keys in a dictionary? You may remember, that hash tables allow us to search for elements in constant time O(1) which is extremely efficient. Python dictionaries (and sets) implement a hash table by default so their keys need to be hashable.

§2. Hashable and unhashable types

In previous topics, we have discussed which objects in Python are mutable and which are not. Let's revise this information very quickly. Strings and integers are **immutable** because we cannot modify them. Putting it simply, if we write `integer += 5` or `string += "end"`, new objects are created, and these `integer` and `string` variables refer to the new objects, not the initial ones. Sets, lists and dictionaries, on the other hand, are **mutable**, so when we alter them, the same objects are modified.

In Python, built-in **immutable** objects like strings or integers are hashable while **mutable** containers like sets, lists or dictionaries are not. Immutable containers (like tuples) are hashable if their elements are hashable. Don't forget about frozensets: they are also immutable and hashable.

Let's see examples of how it works.

```python
# immutable objects
string = "Python"
integer = 4879

print(string.__hash__())  # 227333825058812235
print(integer.__hash__())  # 4879
print(hash(string))  # 227333825058812235
```

There are two ways to get the hash value of an object: built-in `hash()` function or `__hash__()` method of an object. They are equivalent: in fact, `hash()` function internally calls the `__hash__()` method. If we call any of these two methods several times, we'll still get the same value: it is consistent during the lifetime of an object provided no changes occur.

For unhashable objects, the function `hash()` and the `__hash__()` method throw `TypeError`. The same happens if you try to make them dictionary keys or set members:

```python
name_list = ["Julius Caesar"]

print(hash(name_list))  # TypeError: unhashable type: 'list'
dictionary[name_list] = "This is Julius Caesar"  # TypeError
```

The `__eq__` method allows you to compare two values and check if they are equal. Think of it as the equality operator `==`.

Current topic:

✓ Hashable

Topic depends on:

✓ Immutability
✓ Dictionary Stage 1 6★
✓ Set
✓ Tuple
✓ Exceptions Stage 1 3★
✓ Hash table

Topic is required for:

✓ Operations with tuple

Table of contents:

```python
# immutable objects
question = 'ultimate'
answer = 42

print(answer.__eq__(42))           # True
print(answer.__eq__(13))           # False
print(question.__eq__('age-old'))  # False

# mutable objects
numbers = [0, 1]
similar = (0, 1)
print(numbers.__eq__([0, 1]))   # True
print(numbers.__eq__(similar))  # NotImplemented
```

As shown, the method `__eq__` has a wider application, therefore, you can compare both mutable and immutable objects. The comparison typically results in either `True` or `False`, however, the `NotImplemented` object may crop up in cases when the types of given values cannot be compared.

§3. Immutable containers

Immutable containers are hashable if their elements are hashable because the hash value of a container is calculated using the hash values of its elements. You can see it in the tuple examples below:

```python
# tuple with strings
traffic_light = ("red", "yellow", "green")
print(hash(traffic_light))  # -2348372572745757353

# tuple with lists
rainbow = (["red", "orange", "yellow"], ["green", "blue", "purple"])
print(hash(rainbow))  # TypeError: unhashable type: 'list'
```

For built-in types, the hash value depends on the data stored in the object and not on its identity. This is evident when we have two different objects with the same values. Let's take a look at two tuples:

```python
name1 = ("Monty", "Python")
name2 = ("Monty", "Python")

# id
print(id(name1))  # 4539220360
print(id(name2))  # 4539220424

# hash values
print(hash(name1))  # -2157490067397391360
print(hash(name2))  # -2157490067397391360
```

As you can see, even though these are different objects with different ids, their hash values are the same. This means that in a dictionary these two objects will be considered as one:

```python
dictionary = {}
dictionary[name1] = "This is Monty Python"
dictionary[name2] = "This is also Monty Python"

print(dictionary[name1])  # This is also Monty Python
```

The fact that hash values depend on the data in the object also explains why mutable containers are not hashable. If they were, their hash values would change when they would change within the same lifetime of an object. However, the objects in the dictionaries are searched by their hash values. So, when the hash value of an object changes during the lifetime, the key-value pair is lost to you.

§4. Hashable check

In order to avoid errors, we may want to check if an object is hashable. This can be done with the help of the `collections` module. This module has an abstract base class `Hashable` that we can use. The "hash check", so to speak, is carried out with the `isinstance()` function:

```
from collections.abc import Hashable

obj = ...  # some object
isinstance(obj, Hashable)  # True or False
```

Evidently, for hashable objects, this function will return `True`, otherwise — `False`. Below are some examples:

```
# float
isinstance(3.14, Hashable)  # True

# string
isinstance("3.14", Hashable)  # True

# tuple
isinstance((3.14,), Hashable)  # True

# frozenset
isinstance(frozenset({3.14,}), Hashable)  # True

# dict
isinstance({3.14: "Pi number"}, Hashable)  # False

# list
isinstance([3.14], Hashable)  # False

# set
isinstance({3.14,}, Hashable)  # False
```

All of the built-in immutable types you have learned so far are hashable, and the mutable container types (sets, lists, and dictionaries) are not. So for present purposes, you can think of the characteristics *hashable* and *immutable* as synonymous.

§5. Hashable custom classes

Objects of custom classes are hashable by default. This is because the class `object`, which is the parent class of all custom classes, has both `__hash__()` and `__eq__()` methods. You can define a custom implementation of these methods but there are several tricky parts when it comes to that. We won't be covering them in this topic and, anyway, in most cases, default implementations are enough. One thing we will point out, though, is that unlike built-in types, custom classes have their hash values derived from their id's and not their data.

§6. Summary

To sum it up, dictionaries can only use **hashable** objects as their keys. In Python, an object is hashable if it satisfies the following two conditions:

1. We can calculate its **hash value** (that is, an object has the `__hash__()` method)
2. We can **compare** it to other objects (that is, an object has `__eq__()` method)

The concept of hashable objects is connected to the concept of **(im)mutability**. Even though they are different, a good rule of thumb is that **immutable** objects are hashable (with a few exceptions).