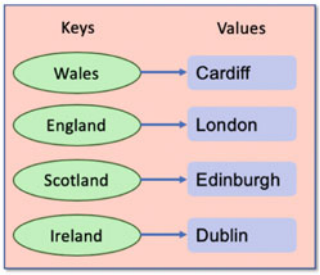
Chapter 33
Dictionaries

33.1 Introduction

A `Dictionary` is a set of associations between a key and a value that is unordered, changeable (mutable) and indexed. Pictorially we might view a Dictionary as shown below for a set of countries and their capital cities. Note that in a Dictionary the keys must be unique but the values do not need to be unique.



33.2 Creating a Dictionary

A `Dictionary` is created using curly brackets (`'{}'`) where each entry in the dictionary is a key:value pair:

```
cities = {'Wales': 'Cardiff',
          'England': 'London',
          'Scotland': 'Edinburgh',
          'Northern Ireland': 'Belfast',
          'Ireland': 'Dublin'}
print(cities)
```


J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_33

This creates a dictionary referenced by the variable `cities` which holds a set of key:value pairs for the Capital cities of the UK and Ireland. When this code is run we see:

```
{'Wales': 'Cardiff', 'England': 'London', 'Scotland':
'Edinburgh', 'Northern Ireland': 'Belfast', 'Ireland':
'Dublin'}
```

33.2.1 The dict() Constructor Function

The `dict()` function can be used to create a new dictionary object from an iterable or a sequence of key:value pairs. The signature of this function is:

```
dict(**kwarg)
dict(mapping, **kwarg)
dict(iterable, **kwarg)
```

This is an overloaded function with three version that can take different types of arguments:

- The first option takes a sequence of key:value pairs.
- The second takes a mapping and (optionally) a sequence of key:value pairs.
- The third version takes an iterable of key:value pairs and an optional sequence of key:value pairs.

Some examples are given below for reference:

```
# note keys are not strings
dict1 = dict(uk='London', ireland='Dublin', france='Paris')
print('dict1:', dict1)
# key value pairs are tuples
dict2 = dict([('uk', 'London'), ('ireland', 'Dublin'),
('france', 'Paris')])
print('dict2:', dict2)
# key value pairs are lists
dict3 = dict((['uk', 'London'], ['ireland', 'Dublin'],
['france', 'Paris']))
print('dict3:', dict3)
```

The output printed by these examples is:

```
dict1: {'uk': 'London', 'ireland': 'Dublin', 'france': 'Paris'}
dict2: {'uk': 'London', 'ireland': 'Dublin', 'france': 'Paris'}
dict3: {'uk': 'London', 'ireland': 'Dublin', 'france': 'Paris'}
```

33.3 Working with Dictionaries

33.3.1 Accessing Items via Keys

You can access the values held in a `Dictionary` using their associated key. This is specified using either the square bracket (`'[]'`) notation (where the key is within the brackets) or the `get()` method:

```
print('cities[Wales]:', cities['Wales'])
print('cities.get(Ireland):', cities.get('Ireland'))
```

The output of this is:

```
cities[Wales]: Cardiff
cities.get(Ireland): Dublin
```

33.3.2 Adding a New Entry

A new entry can be added to a dictionary by providing the key in square brackets and the new value to be assigned to that key:

```
cities['France'] = 'Paris'
```

33.3.3 Changing a Keys Value

The value associated with a key can be changed by reassigning a new value using the square bracket notation, for example:

```
cities['Wales'] = 'Swansea'
print(cities)
```

which would now show 'Swansea' as the capital of wales:

```
{'Wales': 'Swansea', 'England': 'London', 'Scotland':
'Edinburgh', 'Northern Ireland': 'Belfast', 'Ireland':
'Dublin'}
```

33.3.4 Removing an Entry

An entry into the dictionary can be removed using one of the methods `pop()` or `popitem()` method or the `del` keyword.

- The `pop(<key>)` method removes the *entry* with the specified *key*. This method returns the value of the key being deleted. If the key is not present then a default value (if it has been set using `setdefault()`) will be returned. If no default value has been set an error will be generated.
- The `popitem()` method removes the last inserted item in the dictionary (although prior to Python 3.7 a random item in the dictionary was deleted instead!). The *key:value* pair being deleted is returned from the method.
- The `del` keyword removes the entry with the specified key from the dictionary. This keyword just deletes the item; it does not return the associated value. It is potentially more efficient than `pop(<key>)`.

Examples of each of these are given below:

```
cities = {'Wales': 'Cardiff',
          'England': 'London',
          'Scotland': 'Edinburgh',
          'Northern Ireland': 'Belfast',
          'Ireland': 'Dublin'}
print(cities)
cities.popitem() # Deletes 'Ireland' entry
print(cities)
cities.pop('Northern Ireland')
print(cities)
del cities['Scotland']
print(cities)
```

The output from this code snippet is thus:

```
{'Wales': 'Cardiff', 'England': 'London', 'Scotland':
'Edinburgh', 'Northern Ireland': 'Belfast', 'Ireland':
'Dublin'}
{'Wales': 'Cardiff', 'England': 'London', 'Scotland':
'Edinburgh', 'Northern Ireland': 'Belfast'}
{'Wales': 'Cardiff', 'England': 'London', 'Scotland':
'Edinburgh'}
{'Wales': 'Cardiff', 'England': 'London'}
```

In addition the `clear()` method empties the dictionary of all entries:

```
cities = {'Wales': 'Cardiff',
          'England': 'London',
          'Scotland': 'Edinburgh',
          'Northern Ireland': 'Belfast',
          'Ireland': 'Dublin'}
print(cities)
cities.clear()
print(cities)
```

Which generates the following output:

```
{'Wales': 'Cardiff', 'England': 'London', 'Scotland':
'Edinburgh', 'Northern Ireland': 'Belfast', 'Ireland':
'Dublin'}
{}
```

Note that the empty dictionary is represented by the `'{}'` above which as the empty set was represented as `set()`.

33.3.5 Iterating over Keys

You can loop through a dictionary using the `for` loop statement. The `for` loop processes each of the *keys* in the dictionary in turn. This can be used to access each of the values associated with the keys, for example:

```
for country in cities:
    print(country, end=', ')
    print(cities[country])
```

Which generates the output:

```
Wales, Cardiff
England, London
Scotland, Edinburgh
Northern Ireland, Belfast
Ireland, Dublin
```

If you want to iterate over all the values directly, you can do so using the `values()` method. This returns a collection of all the values, which of course you can then iterate over:

```
for e in d.values():
print(e)
```

33.3.6 Values, Keys and Items

There are three methods that allow you to obtain a view onto the contents of a dictionary, these are `values()`, `keys()` and `items()`.

- The `values()` method returns a view onto the dictionary's values.
- The `keys()` method returns a view onto a dictionary's keys.
- The `items()` method returns a view onto the dictionary's items ((key, value) pairs).

A view provides a dynamic window onto the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

The following code iuses the `cities` dictionaries with these three methods:

```
print(cities.values())
print(cities.keys())
print(cities.items())
```

The output makes it clear that these are all related to a dictionary by indicating that the type is `dict_values`, `dict_keys` or `dict_items` etc:

```
dict_values(['Cardiff', 'London', 'Edinburgh', 'Belfast',
'Dublin'])
dict_keys(['Wales', 'England', 'Scotland', 'Northern Ireland',
'Ireland'])
dict_items([('Wales', 'Cardiff'), ('England', 'London'),
('Scotland', 'Edinburgh'), ('Northern Ireland', 'Belfast'),
('Ireland', 'Dublin')])
```

33.3.7 Checking Key Membership

You can check to see if a key is a member of a dictionary using the `in` syntax (and that it is not in a dictionary using the `not in` syntax), for example:

```
print('Wales' in cities)
print('France' not in cities)
```

Which both print out `True` for the `cities` dictionary.

33.3.8 Obtaining the Length of a Dictionary

Once again, as with other collection classes; you can find out the length of a Dictionary (in terms of its key:value pairs) using the `len()` function.

```
cities = {'Wales': 'Cardiff',
          'England': 'London',
          'Scotland': 'Edinburgh',
          'Northern Ireland': 'Belfast',
          'Ireland': 'Dublin'}
print(len(cities)) # prints 5
```

33.3.9 Nesting Dictionaries

The *key* and *value* in a dictionary must be an object; however, everything in Python is an object and thus anything can be used as a key or a value.

One common pattern is where the *value* in a dictionary is itself a container such as a `List`, `Tuple`, `Set` or even another `Dictionary`.

The following example uses Tuples to represent the months that make up the seasons:

```
seasons = {'Spring': ('Mar', 'Apr', 'May'),
           'Summer': ('June', 'July', 'August'),
           'Autumn': ('September', 'October', 'November'),
           'Winter': ('December', 'January', 'February')}
print(seasons['Spring'])
print(seasons['Spring'][1])
```

The output is:

```
('Mar', 'Apr', 'May')
Apr
```

Each season has a `Tuple` for the value element of the entry. When this `Tuple` is returned using the key it can be treated just like any other `Tuple`.

Note in this case we could easily have used a `List` or indeed a `Set` instead of a `Tuple`.

33.4 A Note on Dictionary Key Objects

A class whose objects are to be used as the key within a dictionary should consider implementing two special methods, these are `__hash__()` and `__eq__()`. The hash method is used to generate a hash number that can be used by the dictionary container and the equals method is used to test if two objects are equal. For example:

```
print('key.__hash__():', key.__hash__())
print("key.__eq__('England'):", key.__eq__('England'))
```

The output from these two lines for an example run is:

```
key.__hash__(): 8507681174485233653
key.__eq__('England'): True
```

Python has two rules associated with these methods:

- If two objects are equal, then their hashes should be equal.
- In order for an object to be hashable, it must be immutable.

It also has two properties associated with the hashcodes of an object that should be adhered to:

- If two objects have the same hash, then they are likely to be the same object.
- The hash of an object should be cheap to compute.

Why do you need to care about these methods?

For built in type you do not need to worry; however for user defined c lasses/ types then if these types are to be used as keys within a dictionary then you should consider implementing these methods.

This is because a Dictionary uses

- the *hashing* method to manage how values are organised and
- the *equals* method to check to see if a key is already present in the dictionary.

As an aside if you want to make a class something that cannot be used as a key in a dictionary, that is it is not hashable, then you can define this by setting the `__hash__()` method to `None`.

```
class NotHashableThing(object):
    __hash__ = None
```

33.5 Dictionary Methods

Python has a set of built-in methods that you can use on dictionaries.

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and values
get()	Returns the value of the specified key
items()	Returns a list containing the tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

33.6 Online Resources

Online resources on dictionaries are listed below:

- https://docs.python.org/3/tutorial/datastructures.html Python Tutorial on data structures.
- https://www.python-course.eu/python3_dictionaries.php A tutorial on dictionaries in Python.
- https://docs.python.org/3/library/stdtypes.html#mapping-types-dict for Dictionaries.
- https://docs.python.org/3/tutorial/datastructures.html#dictionaries the online dictionary tutorial.
- https://en.wikipedia.org/wiki/Hash_table For more information on defining hash functions and how they are used in containers such as Dictionary.

33.7 Exercises

The aim of this exercise is to use a `Dictionary` as a simple form of data cache.

Calculating the factorial for a very large number can take some time. For example calculating the factorial of 150,000 can take several seconds. We can verify this using a timer decorator similar to that we created back in chapter on Decorators.

The following program runs several factorial calculations on large numbers and prints out the time taken for each:

```
from timeit import default_timer

def timer(func):
    def inner(value):
        print('calling ', func.__name__, 'with', value)
        start = default_timer()
        func(value)
        end = default_timer()
        print('returned from ', func.__name__, 'it took',
int(end - start), 'seconds')

    return inner

@timer
def factorial(num):
    if num == 0:
        return 1
    else:
        Factorial_value = 1
        for i in range(1, num + 1):
            Factorial_value = factorial_value * i
        return factorial_value

print(factorial(150000))
print(factorial(80000))
print(factorial(120000))
print(factorial(150000))
print(factorial(120000))
print(factorial(80000))
```

An example of the output generated by this program is given below:

```
calling  factorial with 150000
returned from  factorial it took 5 seconds
None
calling  factorial with 80000
returned from  factorial it took 1 seconds
None
calling  factorial with 120000
returned from  factorial it took 3 seconds
None
calling  factorial with 150000
returned from  factorial it took 5 seconds
None
calling  factorial with 120000
returned from  factorial it took 3 seconds
None
calling  factorial with 80000
returned from  factorial it took 1 seconds
None
```

As can be seen from this, in this particular run, calculating the factorial of 150,000 took 5 s, while the factorial of 80,000 took just over 1 1/4 s etc.

In this particular case we have decided to re run these calculations so that we have actually calculated the factorial of 150,000, 80,000 and 120,000 at least twice.

The idea of a *cache* is that it can be used to save previous calculations and reuse those if appropriate rather than have to perform the same calculation multiple times. The use of a cache can greatly improve the performance of systems in which these repeat calculations occur.

There are many commercial caching libraries available for a wide variety of languages including Python. However, at their core they are all somewhat dictionary like; that is there is a *key* which is usually some combination of the operation invoked and the parameter values used. In turn the *value* element is the result of the calculation.

These caches usually also have eviction policies so that they do not become overly large; these eviction policies can usually be specified so that they match the way in which the cache is used. One common eviction policy is the Least Recently Used (or LRU) policy. When using this policy once the size of the cache reaches a predetermined limit the Least Recently Used value is evicted etc.

For this exercise you should implement a simple caching mechanism using a dictionary (but without an eviction policy).

The cache should use the parameter passed into the `factorial()` function as the key and return the stored value if one is present.

The logic for this is usually:

1. Look in the cache to see if the key is present
2. If it is return the value
3. If not perform the calculation
4. Store the calculated result for future use
5. Return the value

Note as the `factorial()` function is exactly that a function; you will need to think about using a global variable to hold the cache.

Once the cache is used with the `factorial()` function, then each subsequent invocation of the function using a previous value should return almost immediately. This is shown in the sample output before where subsequent method calls return in less than a second.

```
calling  factorial with 150000
returned from  factorial it took 5 seconds
None
calling  factorial with 80000
returned from  factorial it took 1 seconds
None
calling  factorial with 120000
returned from  factorial it took 3 seconds
None
calling  factorial with 150000
returned from  factorial it took 0 seconds
None
calling  factorial with 120000
returned from  factorial it took 0 seconds
None
calling  factorial with 80000
returned from  factorial it took 0 seconds
None
```