Topic covered in this python tutorial

1. Mutable or immutable?

A first fundamental distinction that Python makes on data is about whether or not the value of an object changes. If the value can change, the object is called mutable, while if the value cannot change, the object is called immutable.

List of Mutable and Immutable objects

Objects of built-in type that are mutable are:

1. Lists
2. Sets
3. Dictionaries
4. User-Defined Classes (It purely depends upon the user to define the characteristics)

Objects of built-in type that are immutable are:

1. Numbers (Integer, Rational, Float, Decimal, Complex & Booleans)
2. Strings
3. Tuples
4. Frozen Sets
5. User-Defined Classes (It purely depends upon the user to define the characteristics)

Example of Mutable object -

In [1]:

```
# Make a List to save the name of the city
List_cities = ["Delhi", "Mumbai", "Kolkata"]
print("Print the id of List_cities =",id(List_cities))
```

```
Print the id of List_cities = 746962792640
```

In [2]:

```
# Add one more city to the list
List_cities.append("poznan")
print("Print new list :",List_cities)
print("Print the id of List_cities again =",id(List_cities))
```

```
Print new list : ['Delhi', 'Mumbai', 'Kolkata', 'poznan']
Print the id of List_cities again = 746962792640
```

The above example shows us that we were able to change the internal state of the object 'List_cities' by adding one more city 'poznan' to it, yet, the memory address of the object did not change. This confirms that we did not create a new object, rather, the same object was changed or mutated. Hence, we can say that the object which is a type of list with reference variable name 'List_cities' is a MUTABLE OBJECT.

Example of immutable object -

In [3]:

```
# Create a variable "age"
age = 42
print("Id of the age variable :",id(age))
```

```
Id of the age variable : 140720016276544
```

In [4]:

```
# Let's the change the variable value
age = 43
print("Id of the changed 'age' variable :",id(age))
```

```
Id of the changed 'age' variable : 140720016276576
```

In conclusion the id of the age variable is changed so Number objects are immutable.

2. Numbers

Numbers are immutable objects.

2.1 Integers

Integer numbers can be positive, negative, and 0 (zero). They support all the basic mathematical operations, as shown in the following example:

In [5]:

```
a = 14
b = 3

# addition
addition= a + b
print("sum of a+b :",addition)

# subtraction
subtraction=  a - b
print("subtraction of a-b :",subtraction)

# multiplication
multiplication = a * b
print("multiplication of a*b :",multiplication)

# true division
true_division = a / b
print("true division of a/b :",true_division)

# integer division
integer_division = a // b
print("integer division of a//b :",integer_division)

# modulo operation (reminder of division)
modulo_operation=a % b
print("modulo operation of a%b :",modulo_operation)

# power operation
power_operation = a ** b
print("power operation :",power_operation)
```

```
sum of a+b : 17
subtraction of a-b : 11
multiplication of a*b : 42
true division of a/b : 4.666666666666667
integer division of a//b : 4
modulo operation of a%b : 2
power operation : 2744
```

Note:
Python has two division operators, one performs the so-called true division (/), which returns the quotient of the operands, and the other one, the so-called integer division (//), which returns the floored quotient of the operands.

In [6]:

```
# true division
print(" 7 / 4: ",(7 / 4))

# integer division, truncation returns 1
print(" 7 // 4: ",(7 // 4))

# true division again, result is opposite of previous
print(" -7 / 4: ",(-7 / 4))

# integer div., result not the opposite of previous
print(" -7 // 4: ",(-7 // 4))
```

```
 7 / 4:  1.75
 7 // 4:  1
 -7 / 4:  -1.75
 -7 // 4:  -2
```

The result of an integer division in Python is always rounded towards minus infinity. If, instead of flooring, you want to truncate a number to an integer, you can use the built-in int function, as shown in the following example:

In [7]:

```
print(int(-1.75))
print(int(1.75))
```

```
-1
1
```

Notice that the truncation is done toward 0

One nice feature introduced in Python 3.6 is the ability to add underscores within number literals (between digits or base specifiers, but not leading or trailing). The purpose is to help make some numbers more readable, like for example 1_000_000_000:

In [8]:

```
phone_number = 555_666_888
print(phone_number)
```

```
555666888
```

2.2 Booleans

Boolean algebra is that subset of algebra in which the values of the variables are the truth values: true and false. In Python, True and False are two keywords that are used to represent truth values. Booleans are a subclass of integers, and behave respectively like 1 and 0

In [9]:

```
# True behaves like 1
print("int(True) :",int(True))

# False behaves like 0
print("int(False) :",int(False))

# 1 evaluates to True in a boolean context
print("bool(1) :",bool(1))

# and so does every non-zero number
print("bool(-42) :",bool(-42))

# 0 evaluates to False
print("bool(0) :",bool(0))

# quick peak at the operators (and, or, not)
print("not True :",not True)

print("not False :",not False)

print("True and True :",True and True)

print("False or True :",False or True)

print("1 + True :",1 + True)

print("False + 42 :",False + 42)

print("7 - True :",7 - True)
```

```
int(True) : 1
int(False) : 0
bool(1) : True
bool(-42) : True
bool(0) : False
not True : False
not False : True
True and True : True
False or True : True
1 + True : 2
False + 42 : 42
7 - True : 6
```

2.3 Real numbers

Real numbers, or floating point numbers, are represented in Python according to the IEEE 754 double-precision binary floating-point format, which is stored in 64 bits of information divided into three sections: sign, exponent, and mantissa.

Usually, programming languages give coders two different formats: single and double precision. The former takes up 32 bits of memory, and the latter 64. Python supports only the double format. Let's see a simple example:

In [10]:

```
# Let's define pi and radius values

pi = 3.1415926536
radius = 4.5
area = pi * (radius ** 2)
print(area)
```

```
63.617251235400005
```

2.4 Complex Numbers

In [11]:

```
# Define a complex number
c = 3.14 + 2.73j

# real part
print("c.real :",c.real)

# imaginary part
print("c.imag :",c.imag)


# conjugate of A + Bj is A - Bj
print("c.conjugate() :",c.conjugate())


# multiplication is allowed
print("c * 2  :",c * 2)


# power operation as well
print("c ** 2 :",c ** 2)


# addition and subtraction as well
d = 1 + 1j
print("c - d :",c - d)
```

```
c.real : 3.14
c.imag : 2.73
c.conjugate() : (3.14-2.73j)
c * 2  : (6.28+5.46j)
c ** 2 : (2.4067000000000007+17.1444j)
c - d : (2.14+1.73j)
```

2.5 Fractions and decimals

Fractions hold a rational numerator and denominator in their lowest forms. Let's see a quick example:

In [21]:

```
from fractions import Fraction
A = Fraction(10, 6)  # mad hatter?
print(A)  # notice it's been simplified after printing

print(Fraction(1, 3) + Fraction(2, 3))  # 1/3 + 2/3 == 3/3 == 1/1

f = Fraction(10, 6)
print("f numerator :",f.numerator)
print("f denominator :",f.denominator)
```

```
5/3
1
f numerator : 5
f denominator : 3
```

Let's see a quick example with decimal numbers:

In [23]:

```
from decimal import Decimal as D  # rename for brevity
a = D(3.14)  # pi, from float, so approximation issues
print(a)

b=D('3.14')  # pi, from a string, so no approximation issues
print(b)

c = D(0.1) * D(3) - D(0.3)  # from float, we still have the issue
print(c)

d= D('0.1') * D(3) - D('0.3')  # from string, all perfect
print(d)

e = D('1.4').as_integer_ratio()  # 7/5 = 1.4 (isn't this cool?!)
print(e)
```

```
3.140000000000000124344978758017532527446746826171875
3.14
2.77555756156515654042363166 8E-17
0.0
(7, 5)
```

Notice that when we construct a Decimal number from a float, it takes on all the approximation issues float may come from. On the other hand, when the Decimal has no approximation issues (for example, when we feed an int or a string representation to the constructor), then the calculation has no quirky behavior. When it comes to money, use decimals.

3. Strings

Let's start with immutable sequences: strings, tuples, and bytes

Textual data in Python is handled with str objects, more commonly known as strings. They are immutable sequences of Unicode code points. Unicode code points can represent a character, but can also have other meanings, such as formatting data, for example. Python, unlike other languages, doesn't have a char type, so a single character is rendered simply by a string of length 1

Unicode is an excellent way to handle data, and should be used for the internals of any application. When it comes to storing textual data though, or sending it on the network, you may want to encode it, using an appropriate encoding for the medium you're using. The result of an encoding produces a bytes object, whose syntax and behavior is similar to that of strings. String literals are written in Python using single, double, or triple quotes (both single or double). If built with triple quotes, a string can span on multiple lines. An example will clarify this:

In [25]:

```
# 4 ways to make a string
str1 = 'This is a string. We built it with single quotes.'
str2 = "This is also a string, but built with double quotes."
str3 = '''This is built using triple quotes,
so it can span multiple lines.'''
str4 = """This too
is a multiline one
built with triple double-quotes."""
str5='This too\nis a multiline one\nbuilt with triple double-quotes.'

print(str1)
print(" ")
print(str2)
print(" ")
print(str3)
print(" ")
print(str4)
print(" ")
print(str5)
```

```
This is a string. We built it with single quotes.

This is also a string, but built with double quotes.

This is built using triple quotes,
so it can span multiple lines.

This too
is a multiline one
built with triple double-quotes.

This too
is a multiline one
built with triple double-quotes.
```

Strings, like any sequence, have a length. You can get this by calling the len function:

In [26]:

```
print(len(str1))
```

```
49
```

4. Encoding and decoding strings

Using the encode/decode methods, we can encode Unicode strings and decode bytes objects. UTF-8 is a variable length character encoding, capable of encoding all possible Unicode code points. It is the dominant encoding for the web. Notice also that by adding a literal b in front of a string declaration, we're creating a bytes object:

In [36]:

```
s = "This is üŋíc0de"  # unicode string: code points
print(type(s))

encoded_s = s.encode('utf-8')  # utf-8 encoded version of s
print(encoded_s)  # result: bytes object

print(type(encoded_s))  # another way to verify it

decode = encoded_s.decode('utf-8')  # let's revert to the original
print(decode)

bytes_obj = b"A bytes object"  # a bytes object
print(type(bytes_obj))
```

```
<class 'str'>
b'This is \xc3\xbc\xc5\x8b\xc3\xadc0de'
<class 'bytes'>
This is üŋíc0de
<class 'bytes'>
```

PNG, JPEG, MP3, WAV, ASCII, UTF-8 etc are different forms of encodings. An encoding is a format to represent audio, images, text, etc in bytes. Converting Strings to byte objects is termed as encoding. This is necessary so that the text can be stored on disk using mapping using ASCII or UTF-8 encoding techniques. This task is achieved using encode(). It take encoding technique as argument. Default technique is “UTF-8” technique.

In [37]:

```
# Python code to demonstrate String encoding

# initialising a String
a = 'GeeksforGeeks'

# initialising a byte object
c = b'GeeksforGeeks'

# using encode() to encode the String
# encoded version of a is stored in d
# using ASCII mapping
d = a.encode('ASCII')

# checking if a is converted to bytes or not
if (d==c):
        print ("Encoding successful")
else : print ("Encoding Unsuccessful")
```

```
Encoding successful
```

5. Indexing and slicing strings

When manipulating sequences, it's very common to have to access them at one precise position (indexing), or to get a subsequence out of them (slicing). When dealing with immutable sequences, both operations are read-only.

While indexing comes in one form, a zero-based access to any position within the sequence, slicing comes in different forms. When you get a slice of a sequence, you can specify the start and stop positions, and the step. They are separated with a colon (:) like this: my_sequence[start:stop:step]. All the arguments are optional, start is inclusive, and stop is exclusive. It's much easier to show an example, rather than explain them further in words:

In [29]:

```
s = "The trouble is you think you have time."
print(s[0])  # indexing at position 0, which is the first char

print(s[5])  # indexing at position 5, which is the sixth char

print(s[:4])  # slicing, we specify only the stop position

print(s[4:])  # slicing, we specify only the start position

print(s[2:14])  # slicing, both start and stop positions

print(s[2:14:3])  # slicing, start, stop and step (every 3 chars)

print(s[:])  # quick way of making a copy
```

```
T
r
The
trouble is you think you have time.
e trouble is
erb
The trouble is you think you have time.
```

In [30]:

```
a = list(range(10))  # `a` has 10 elements. Last one is 9.
print(a)

print(len(a))  # its length is 10 elements

print(a[len(a) - 1])  # position of last one is len(a) - 1

print(a[-1])  # but we don't need len(a)! Python rocks!

print(a[-2])  # equivalent to len(a) - 2

print(a[-3])  # equivalent to len(a) - 3
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
10
9
9
8
7
```

6. String formatting

One of the features strings have is the ability to be used as a template. There are several different ways of formatting a string, and for the full list of possibilities. Here are some common examples:

In [31]:

```
greet_old = 'Hello %s!'
b= greet_old % 'Fabrizio'
print(b)

greet_positional = 'Hello {} {}!'
c=greet_positional.format('Fabrizio', 'Romano')
print(c)

greet_positional_idx = 'This is {0}! {1} loves {0}!'
d=greet_positional_idx.format('Python', 'Fabrizio')
print(d)
e=greet_positional_idx.format('Coffee', 'Fab')
print(e)

keyword = 'Hello, my name is {name} {last_name}'
f=keyword.format(name='Fabrizio', last_name='Romano')
print(f)
```

```
Hello Fabrizio!
Hello Fabrizio Romano!
This is Python! Fabrizio loves Python!
This is Coffee! Fab loves Coffee!
Hello, my name is Fabrizio Romano
```

7. Tuples

A tuple is a sequence of arbitrary Python objects. In a tuple, items are separated by commas. They are used everywhere in Python, because they allow for patterns that are hard to reproduce in other languages. Sometimes tuples are used implicitly; for example, to set up multiple variables on one line, or to allow a function to return multiple different objects (usually a function returns one object only, in many other languages), and even in the Python console, you can use tuples implicitly to print multiple elements with one single instruction. We'll see examples for all these cases:

In [32]:

```
t = ()  # empty tuple
print(type(t))

one_element_tuple = (42, )  # you need the comma!
three_elements_tuple = (1, 3, 5)  # braces are optional here

print(one_element_tuple)
print(three_elements_tuple)

a, b, c = 1, 2, 3  # tuple for multiple assignment

print(a, b, c)  # implicit tuple to print with one instruction

print(3 in three_elements_tuple)  # membership test
```

```
<class 'tuple'>
(42,)
(1, 3, 5)
1 2 3
True
```

Notice that the membership operator in can also be used with lists, strings, dictionaries, and, in general, with collection and sequence objects.

Notice that to create a tuple with one item, we need to put that comma after the item. The reason is that without the comma that item is just itself wrapped in braces, kind of in a redundant mathematical expression. Notice also that on assignment, braces are optional so my_tuple = 1, 2, 3 is the same as my_tuple = (1, 2, 3).

One thing that tuple assignment allows us to do, is one-line swaps, with no need for a third temporary variable. Let's see first a more traditional way of doing it:

In [33]:

```
a, b = 1, 2
c = a  # we need three lines and a temporary var c
a = b
b = c
print(a, b)  # a and b have been swapped
```

```
2 1
```

And now let's see how we would do it in Python:

In [34]:

```
a, b = 0, 1
a, b = b, a  # this is the Pythonic way to do it
print(a, b)
```

```
1 0
```

8. Lists

Mutable sequences differ from their immutable sisters in that they can be changed after creation. There are two mutable sequence types in Python: lists and byte arrays. I said before that the dictionary is the king of data structures in Python. I guess this makes the list its rightful queen.

Python lists are mutable sequences. They are very similar to tuples, but they don't have the restrictions of immutability. Lists are commonly used to storing collections of homogeneous objects, but there is nothing preventing you from store heterogeneous collections as well. Lists can be created in many different ways. Let's see an example:

In [39]:

```
list_type1 = []
print(type(list_type1))

list_type2 = list()  # same as []
print(type(list_type2))

list_type3 = [1, 2, 3]  # as with tuples, items are comma separated
print(type(list_type3))

list_type4 = [x + 5 for x in [2, 3, 4]]  # Python is magic
print(type(list_type4))
print(list_type4)

list_type5 =list((1, 3, 5, 7, 9))  # list from a tuple
print(type(list_type5))
print(list_type5)

list_type6 =list('hello')  # list from a string
print(type(list_type6))
print(list_type6)

print(list_type6[0])
```

```
<class 'list'>
<class 'list'>
<class 'list'>
<class 'list'>
[7, 8, 9]
<class 'list'>
[1, 3, 5, 7, 9]
<class 'list'>
['h', 'e', 'l', 'l', 'o']
o
```

Operations with List

In [40]:

```
a = [1, 2, 1, 3]
print(a)

a.append(13)  # we can append anything at the end
print(a)

print(a.count(1))  # how many `1` are there in the list?

a.extend([5, 7])  # extend the list by another (or sequence)
print(a)

print(a.index(13))  # position of `13` in the list (0-based indexing)


a.insert(0, 17)  # insert `17` at position 0
print(a)

a.pop()  # pop (remove and return) last element
print(a)

a.pop(3)  # pop element at position 3
print(a)


a.remove(17)  # remove `17` from the list
print(a)

a.reverse()  # reverse the order of the elements in the list
print(a)

a.sort()  # sort the list
print(a)

a.clear()  # remove all elements from the list
print(a)
```

```
[1, 2, 1, 3]
[1, 2, 1, 3, 13]
2
[1, 2, 1, 3, 13, 5, 7]
4
[17, 1, 2, 1, 3, 13, 5, 7]
[17, 1, 2, 1, 3, 13, 5]
[17, 1, 2, 3, 13, 5]
[1, 2, 3, 13, 5]
[5, 13, 3, 2, 1]
[1, 2, 3, 5, 13]
[]
```

You can extend lists using any sequence type:

In [41]:

```
a = list('hello')  # makes a list from a string
print(a)

a.append(100)  # append 100, heterogeneous type
print(a)

a.extend((1, 2, 3))  # extend using tuple
print(a)

a.extend('...')  # extend using string
print(a)
```

```
['h', 'e', 'l', 'l', 'o']
['h', 'e', 'l', 'l', 'o', 100]
['h', 'e', 'l', 'l', 'o', 100, 1, 2, 3]
['h', 'e', 'l', 'l', 'o', 100, 1, 2, 3, '.', '.', '.']
```

Now, let's see what are the most common operations you can do with lists:

In [42]:

```
a = [1, 3, 5, 7]
print(min(a))  # minimum value in the list

print(max(a))  # maximum value in the list

print(sum(a))  # sum of all values in the list

print(len(a))  # number of elements in the list

b = [6, 7, 8]
print(a + b)

print(a * 2)  # `*` has also a special meaning
```

```
1
7
16
4
[1, 3, 5, 7, 6, 7, 8]
[1, 3, 5, 7, 1, 3, 5, 7]
```

The last two lines in the preceding code are quite interesting because they introduce us to a concept called operator overloading. In short, it means that operators such as +, -. , *%, and so on, may represent different operations according to the context they are used in. It doesn't make any sense to sum two lists, right? Therefore, the + sign is used to concatenate them. Hence, the* sign is used to concatenate the list to itself according to the right operand.

9. Byte arrays

To conclude our overview of mutable sequence types, let's spend a couple of minutes on the bytearray type. Basically, they represent the mutable version of bytes objects. They expose most of the usual methods of mutable sequences as well as most of the methods of the bytes type. Items are integers in the range [0, 256).

In [47]:

```
a = bytearray()  # empty bytearray object
print(a)

b = bytearray(10)  # zero-filled instance with given length
print(b)

print(range(5)) # This is how the range function works

c = bytearray(range(5)) # bytearray from iterable of integers
print(c)

name = bytearray(b'Lina')  #A - bytearray from bytes
print(name)
print(name.replace(b'L', b'l'))


print(name.endswith(b'na'))

print(name.upper())

print(name.count(b'L'))
```

```
bytearray(b'')
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
range(0, 5)
bytearray(b'\x00\x01\x02\x03\x04')
bytearray(b'Lina')
bytearray(b'lina')
True
bytearray(b'LINA')
1
```

As you can see in the preceding code, there are a few ways to create a bytearray object. They can be useful in many situations; for example, when receiving data through a socket, they eliminate the need to concatenate data while polling, hence they can prove to be very handy. On the line #A, I created a bytearray named as name from the bytes literal b'Lina' to show you how the bytearray object exposes methods from both sequences and strings, which is extremely handy. If you think about it, they can be considered as mutable strings.

10. Set types

Python also provides two set types, set and frozenset. The set type is mutable, while frozenset is immutable. They are unordered collections of immutable objects. Hashability is a characteristic that allows an object to be used as a set member as well as a key for a dictionary, as we'll see very soon.

Objects that compare equally must have the same hash value. Sets are very commonly used to test for membership, so let's introduce the in operator in the following example:

In [48]:

```
small_primes = set()  # empty set
small_primes.add(2)  # adding one element at a time
small_primes.add(3)
small_primes.add(5)
print(small_primes)

small_primes.add(1)  # Look what I've done, 1 is not a prime!
print(small_primes)

small_primes.remove(1)  # so let's remove it
print(small_primes)

print(3 in small_primes) # membership test

print(4 in small_primes)

print(4 not in small_primes)  # negated membership test

small_primes.add(3)  # trying to add 3 again
print(small_primes)# no change, duplication is not allowed

bigger_primes = set([5, 7, 11, 13])  # faster creation
print(small_primes | bigger_primes) # union operator `|`

print(small_primes & bigger_primes)  # intersection operator `&`

print(small_primes - bigger_primes)  # difference operator `-`
```

```
{2, 3, 5}
{1, 2, 3, 5}
{2, 3, 5}
True
False
True
{2, 3, 5}
{2, 3, 5, 7, 11, 13}
{5}
{2, 3}
```

Another way of creating a set is by simply using the curly braces notation, like this:

In [49]:

```
small_primes = {2, 3, 5, 5, 3}
print(small_primes)
```

```
{2, 3, 5}
```

Notice I added some duplication to emphasize that the resulting set won't have any. Let's see an example about the immutable counterpart of the set type, frozenset:

In [50]:

```
small_primes = frozenset([2, 3, 5, 7])
bigger_primes = frozenset([5, 7, 11])

print(small_primes.add(11)) # we cannot add to a frozenset
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call las
t)
<ipython-input-50-3c72a5d7c252> in <module>
      2 bigger_primes = frozenset([5, 7, 11])
      3
----> 4 print(small_primes.add(11)) # we cannot add to a frozenset
      5 print(small_primes.remove(2))  # neither we can remove
      6

AttributeError: 'frozenset' object has no attribute 'add'
```

In [51]:

```
print(small_primes.remove(2))  # neither we can remove
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call las
t)
<ipython-input-51-f09751d16dca> in <module>
----> 1 print(small_primes.remove(2))  # neither we can remove

AttributeError: 'frozenset' object has no attribute 'remove'
```

In [52]:

```
print(small_primes & bigger_primes)  # intersect, union, etc. allowed
```

```
frozenset({5, 7})
```

11. Dictionaries

Of all the built-in Python data types, the dictionary is easily the most interesting one. It's the only standard mapping type, and it is the backbone of every Python object.

A dictionary maps keys to values. Keys need to be hashable objects, while values can be of any arbitrary type. Dictionaries are mutable objects. There are quite a few different ways to create a dictionary, so let me give you a simple example of how to create a dictionary equal to {'A': 1, 'Z': -1} in five different ways

In [1]:

```
a = dict(A=1, Z=-1)
b = {'A': 1, 'Z': -1}
c = dict(zip(['A', 'Z'], [1, -1]))
d = dict([('A', 1), ('Z', -1)])
e = dict({'Z': -1, 'A': 1})
print(a == b == c == d == e)  # are they all the same? # They are indeed
```

```
True
```

Have you noticed those double equals? Assignment is done with one equal, while to check whether an object is the same as another one (or five in one go, in this case), we use double equals. There is also another way to compare objects, which involves the is operator, and checks whether the two objects are the same (if they have the same ID, not just the value), but unless you have a good reason to use it, you should use the double equals instead. In the preceding code, I also used one nice function: zip. It is named after the real-life zip, which glues together two things taking one element from each at a time. Let me show you an example:

In [2]:

```
a =list(zip(['h', 'e', 'l', 'l', 'o'], [1, 2, 3, 4, 5]))
b =list(zip('hello', range(1, 6)))  # equivalent, more Pythonic
print(a)
print("")
print(b)
```

```
[('h', 1), ('e', 2), ('l', 3), ('l', 4), ('o', 5)]

[('h', 1), ('e', 2), ('l', 3), ('l', 4), ('o', 5)]
```

In the preceding example, I have created the same list in two different ways, one more explicit, and the other a little bit more Pythonic. Forget for a moment that I had to wrap the list constructor around the zip call (the reason is because zip returns an iterator, not a list, so if I want to see the result I need to exhaust that iterator into something—a list in this case), and concentrate on the result. See how zip has coupled the first elements of its two arguments together, then the second ones, then the third ones, and so on and so forth? Take a look at your pants (or at your purse, if you're a lady) and you'll see the same behavior in your actual zip. But let's go back to dictionaries and see how many wonderful methods they expose for allowing us to manipulate them as we want.

Let's start with the basic operations:

In [3]:

```
d = {}
d['a'] = 1  # let's set a couple of (key, value) pairs
d['b'] = 2
print(len(d))  # how many pairs?

print(d['a'])  # what is the value of 'a'?

print(d)  # how does `d` look now?

del d['a']  # let's remove `a`
print(d)

d['c'] = 3  # let's add 'c': 3
print('c' in d)  # membership is checked against the keys

print(3 in d)  # not the values

print('e' in d)

d.clear()  # let's clean everything from this dictionary
print(d)
```

```
2
1
{'a': 1, 'b': 2}
{'b': 2}
True
False
False
{}
```

Notice how accessing keys of a dictionary, regardless of the type of operation we're performing, is done through square brackets. Do you remember strings, lists, and tuples? We were accessing elements at some position through square brackets as well, which is yet another example of Python's consistency.

Let's see now three special objects called dictionary views: keys, values, and items. These objects provide a dynamic view of the dictionary entries and they change when the dictionary changes. keys() returns all the keys in the dictionary, values() returns all the values in the dictionary, and items() returns all the (key, value) pairs in the dictionary.

According to the Python documentation: "Keys and values are iterated over in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary's history of insertions and deletions. If keys, values and items views are iterated over with no intervening modifications to the dictionary, the order of items will directly correspond." Enough with this chatter; let's put all this down into code:

In [4]:

```
d = dict(zip('hello', range(5)))
print(d)

print(d.keys())

print(d.values())

print(d.items())

print(3 in d.values())

print(('o', 4) in d.items())
```

```
{'h': 0, 'e': 1, 'l': 3, 'o': 4}
dict_keys(['h', 'e', 'l', 'o'])
dict_values([0, 1, 3, 4])
dict_items([('h', 0), ('e', 1), ('l', 3), ('o', 4)])
True
True
```

There are a few things to notice in the preceding code. First, notice how we're creating a dictionary by iterating over the zipped version of the string 'hello' and the list [0, 1, 2, 3, 4]. The string 'hello' has two 'l' characters inside, and they are paired up with the values 2 and 3 by the zip function. Notice how in the dictionary, the second occurrence of the 'l' key (the one with value 3), overwrites the first one (the one with value 2). Another thing to notice is that when asking for any view, the original order is now preserved, while before Version 3.6 there was no guarantee of that.

As of Python 3.6, the dict type has been reimplemented to use a more compact representation. This resulted in dictionaries using 20% to 25% less memory when compared to Python 3.5. Moreover, in Python 3.6, as a side effect, dictionaries are natively ordered. This feature has received such a welcome from the community that in 3.7 it has become a legit feature of the language rather than an implementation side effect. A dict is ordered if it remembers the order in which keys were first inserted. We'll see how these views are fundamental tools when we talk about iterating over collections. Let's take a look now at some other methods exposed by Python's dictionaries; there's plenty of them and they are very useful:

In [5]:

```
print(d)

d.popitem()  # removes a random item (useful in algorithms)
print(d)

print(d.pop('l'))  # remove item with key `l`

print(d.pop('not-a-key'))  # remove a key not in dictionary: KeyError
```

```
{'h': 0, 'e': 1, 'l': 3, 'o': 4}
{'h': 0, 'e': 1, 'l': 3}
3

---------------------------------------------------------------------------
-
KeyError                                  Traceback (most recent call las
t)
<ipython-input-5-51e897cdf4c9> in <module>
      6 print(d.pop('l'))  # remove item with key `l`
      7
----> 8 print(d.pop('not-a-key'))  # remove a key not in dictionary: KeyEr
ror
      9

KeyError: 'not-a-key'
```

In [6]:

```
print(d.pop('not-a-key', 'default-value'))  # with a default value?

d.update({'another': 'value'})  # we can update dict this way
d.update(a=13)  # or this way (like a function call)
print(d)
print(d.get('a'))  # same as d['a'] but if key is missing no KeyError

print(d.get('a', 177))  # default value used if key is missing

print(d.get('b', 177))  # like in this case

print(d.get('b'))  # key is not there, so None is returned
```

```
default-value
{'h': 0, 'e': 1, 'another': 'value', 'a': 13}
13
13
177
None
```

All these methods are quite simple to understand, but it's worth talking about that None, for a moment. Every function in Python returns None, unless the return statement is explicitly used to return something else, but we'll see this when we explore functions. None is frequently used to represent the absence of a value, and it is quite commonly used as a default value for arguments in function declaration. Some inexperienced coders sometimes write code that returns either False or None. Both False and None evaluate to False in a Boolean context so it may seem there is not much difference between them. But actually, I would argue there is quite an important difference: False means that we have information, and the information we have is False. None means no information. And no information is very different from information that is False. In layman's terms, if you ask your mechanic, Is my car ready?, there is a big difference between the answer, No, it's not (False) and, I have no idea (None).

One last method I really like about dictionaries is setdefault. It behaves like get, but also sets the key with the given value if it is not there. Let's see an example:

In [7]:

```
d = {}
d.setdefault('a', 1)  # 'a' is missing, we get default value

print(d) # also, the key/value pair ('a', 1) has now been added

print(d.setdefault('a', 5))  # let's try to override the value

print(d)# no override, as expected
```

```
{'a': 1}
1
{'a': 1}
```

12. namedtuple

A namedtuple is a tuple-like object that has fields accessible by attribute lookup as well as being indexable and iterable (it's actually a subclass of tuple). This is sort of a compromise between a full-fledged object and a tuple, and it can be useful in those cases where you don't need the full power of a custom object, but you want your code to be more readable by avoiding weird indexing. Another use case is when there is a chance that items in the tuple need to change their position after refactoring, forcing the coder to refactor also all the logic involved, which can be very tricky. As usual, an example is better than a thousand words (or was it a picture?). Say we are handling data about the left and right eyes of a patient. We save one value for the left eye (position 0) and one for the right eye (position 1) in a regular tuple. Here's how that might be:

In [8]:

```
vision = (9.5, 8.8)
print(vision)

print(vision[0])  # left eye (implicit positional reference)

print(vision[1])  # right eye (implicit positional reference)
```

```
(9.5, 8.8)
9.5
8.8
```

Now let's pretend we handle vision objects all the time, and at some point the designer decides to enhance them by adding information for the combined vision, so that a vision object stores data in this format: (left eye, combined, right eye).

Do you see the trouble we're in now? We may have a lot of code that depends on vision[0] being the left eye information (which it still is) and vision[1] being the right eye information (which is no longer the case). We have to refactor our code wherever we handle these objects, changing vision[1] to vision[2], and it can be painful. We could have probably approached this a bit better from the beginning, by using a namedtuple. Let me show you what I mean:

In [9]:

```
from collections import namedtuple
Vision = namedtuple('Vision', ['left', 'right'])
vision = Vision(9.5, 8.8)
print(vision[0])

print(vision.left)  # same as vision[0], but explicit

print(vision.right)  # same as vision[1], but explicit
```

```
9.5
9.5
8.8
```

You can see how convenient it is to refer to those values by name rather than by position. After all, a wise man once wrote, Explicit is better than implicit (can you recall where? Think Zen if you can't...). This example may be a little extreme; of course, it's not likely that our code designer will go for a change like this, but you'd be amazed to see how frequently issues similar to this one happen in a professional environment, and how painful it is to refactor them.

13. defaultdict

The defaultdict data type is one of my favorites. It allows you to avoid checking if a key is in a dictionary by simply inserting it for you on your first access attempt, with a default value whose type you pass on creation. In some cases, this tool can be very handy and shorten your code a little. Let's see a quick example. Say we are updating the value of age, by adding one year. If age is not there, we assume it was 0 and we update it to 1:

In [10]:

```
d = {}
d['age'] = d.get('age', 0) + 1  # age not there, we get 0 + 1
print(d)

d = {'age': 39}
d['age'] = d.get('age', 0) + 1  # age is there, we get 40
print(d)
```

```
{'age': 1}
{'age': 40}
```

Now let's see how it would work with a defaultdict data type. The second line is actually the short version of a four-lines-long if clause that we would have to write if dictionaries didn't have the get method

In [11]:

```
from collections import defaultdict
dd = defaultdict(int)  # int is the default type (0 the value)
dd['age'] += 1  # short for dd['age'] = dd['age'] + 1
print(dd)
```

```
defaultdict(<class 'int'>, {'age': 1})
```

Notice how we just need to instruct the defaultdict factory that we want an int number to be used in case the key is missing (we'll get 0, which is the default for the int type). Also, notice that even though in this example there is no gain on the number of lines, there is definitely a gain in readability, which is very important. You can also use a different technique to instantiate a defaultdict data type, which involves creating a factory object.

14. ChainMap

ChainMap is an extremely nice data type which was introduced in Python 3.3. It behaves like a normal dictionary but according to the Python documentation: "is provided for quickly linking a number of mappings so they can be treated as a single unit". This is usually much faster than creating one dictionary and running multiple update calls on it. ChainMap can be used to simulate nested scopes and is useful in templating. The underlying mappings are stored in a list. That list is public and can be accessed or updated using the maps attribute. Lookups search the underlying mappings successively until a key is found. By contrast, writes, updates, and deletions only operate on the first mapping.

A very common use case is providing defaults, so let's see an example:

In [12]:

```
from collections import ChainMap
default_connection = {'host': 'localhost', 'port': 4567}
connection = {'port': 5678}
conn = ChainMap(connection, default_connection)  # map creation
print(conn['port'])  # port is found in the first dictionary

print(conn['host'])  # host is fetched from the second dictionary

print(conn.maps)  # we can see the mapping objects

conn['host'] = 'packtpub.com'  # let's add host
print(conn.maps)

del conn['port']  # let's remove the port information
print(conn.maps)

print(conn['port'])  # now port is fetched from the second dictionary

print(dict(conn))  # easy to merge and convert to regular dictionary
```

```
5678
localhost
[{'port': 5678}, {'host': 'localhost', 'port': 4567}]
[{'port': 5678, 'host': 'packtpub.com'}, {'host': 'localhost', 'port': 456
7}]
[{'host': 'packtpub.com'}, {'host': 'localhost', 'port': 4567}]
4567
{'host': 'packtpub.com', 'port': 4567}
```

15. Enums

Technically not a built-in data type, as you have to import them from the enum module, but definitely worth mentioning, are enumerations. They were introduced in Python 3.4, and though it is not that common to see them in professional code (yet), I thought I'd give you an example anyway.

The official definition goes like this: "An enumeration is a set of symbolic names (members) bound to unique, constant values. Within an enumeration, the members can be compared by identity, and the enumeration itself can be iterated over."

Say you need to represent traffic lights. In your code, you might resort to doing this:

In [13]:

```
GREEN = 1
YELLOW = 2
RED = 4
TRAFFIC_LIGHTS = (GREEN, YELLOW, RED)
 # or with a dict
traffic_lights = {'GREEN': 1, 'YELLOW': 2, 'RED': 4}
print(TRAFFIC_LIGHTS)
print(traffic_lights)
```

```
(1, 2, 4)
{'GREEN': 1, 'YELLOW': 2, 'RED': 4}
```

There's nothing special about the preceding code. It's something, in fact, that is very common to find. But, consider doing this instead:

In [14]:

```
from enum import Enum
class TrafficLight(Enum):
    GREEN = 1
    YELLOW = 2
    RED = 4

print(TrafficLight.GREEN)

print(TrafficLight.GREEN.name)

print(TrafficLight.GREEN.value)

print(TrafficLight(1))

print(TrafficLight(4))
```

```
TrafficLight.GREEN
GREEN
1
TrafficLight.GREEN
TrafficLight.RED
```

Ignoring for a moment the (relative) complexity of a class definition, you can appreciate how this might be more advantageous. The data structure is much cleaner, and the API it provides is much more powerful. I encourage you to check out the official documentation to explore all the great features you can find in the enum module. I think it's worth exploring, at least once.

In [ ]:

```

```