Intro to Python
Lab 3

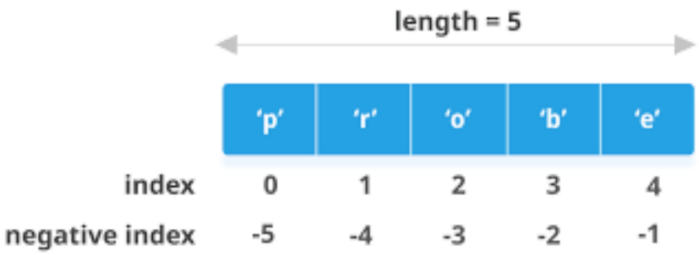
1 Lists

File to use: `list-basics.py`

A list is an ordered collection of items, which may be of the same or different data type. Python uses the square brackets [] to indicate a list. Lists are conceptually equivalent to arrays in other languages, although Python lists offer more functionality and flexibility compared to arrays.

Typically, a list contains one or more items separated by commas.

You access elements in a list based on their index. Lists in Python are zero-indexed, which means the first element in the list has index 0, the second element has an index of 1, and so on.
Negative indexing allows you to access elements starting from the end of the list



As mentioned earlier, EVERYTHING IN PYTHON IS AN OBJECT. A list (along with sets and dictionaries) are mutable objects, while strings, numbers, tuples are immutable. An object whose internal state can be changed is called a mutable object, while an object whose internal state cannot be changed is called an immutable object.

List are iterables. Iterables are Python objects that is capable of returning its contents one item at a item (through a for loop). Other examples of iterables are strings, tuples

You can modify elements in the list, add new elements to the list, and remove elements from a list through a variety of methods.

These are some of the more common methods used with lists:
https://www.programiz.com/python-programming/methods/list

1.1 List slicing

File to use: `list-slicing.py`

Lists support the slice notation which provide a very effective way to manipulate lists in a variety of ways, as well as obtaining a sublist from a larger original list.

```
sub_list = list[begin: end: step]
```

In this syntax, the `begin`, `end`, and `step` arguments must be valid indexes. And they're all optional.

The `begin` index defaults to zero. The `end` index defaults to the length of the list. And the `step` index defaults to 1. The slice will start from the `begin` up to one element before the `end` following `step`.
The `begin`, `end`, and `step` can be positive or negative. Positive values slice the list from the first element to the last element while negative values slice the list from the last element to the first element.

1.2 List operations on a string

File to use: `string-intermediate.py`

A string is basically a list of characters, so the standard operations on a list (indexing, slicing, iteration) can also be applied to a string.
The key difference between a string and a list is that a string is immutable (you cannot change its contents), unlike a list. So, any list operations that modify the content of a list cannot be used on a string.

1.3 Lambda / anonymous functions for comprehensions and higher order functions

File to use: `functions-lambda-anonymous.py`

Lambda / anonymous functions are similar to normal user-defined functions but without a name. Lambda functions are efficient whenever you want to create a function that will only contain simple expressions – that is, expressions that are usually a single line of a statement (no if-else , for-loops, etc). They're also useful when you want to use the function only once (unlike normal functions which are meant to be called repeatedly throughout program execution).

Lambda functions are specifically used with list comprehensions and also higher-order functions (such as map and reduce).

1.4 List comprehensions

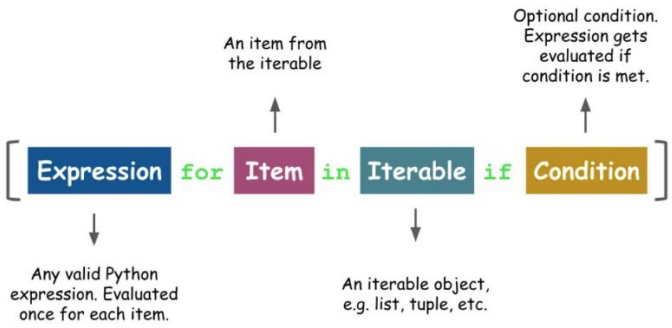
File to use: `list-comprehension.py`

List comprehension is an elegant way to define and create lists based on existing iterables (lists, tuples, strings, etc). It is generally more compact and faster than normal functions and loops for creating lists.

The general syntax for list comprehension looks like this:

```
new_list = [expression for variable in iterable]
```

List comprehensions start and end with opening and closing square brackets, [].
Then comes the `expression` or operation you'd like to perform and carry out on each value inside the current iterable. The results of these calculations enter the new list.
The `expression` is followed by a `for` clause.
`variable` is a temporary name you want to use for each item in the current list that is going through the iteration.
The `in` keyword is used to loop over the iterable: `iterable` can be any Python object, such as a list, tuple, string and so on.
From the iteration that was performed and the calculations that took place on each item during the iteration, new values were created which are saved to a variable, in this case `new_list`. The old list (or other object) will remain unchanged.
There can be an optional `if` statement and additional `for` clause.



Nested list comprehension is when this process is run on a list of lists where each nested list and element is operated upon.

1.5 map and filter higher order functions

File to use: `map-filter.py`

When working with a list (or a tuple), you often need to transform or filter the elements of the list and return a new list. This can be done with a loop in a conventional way, but the built-in higher order functions map, filter and reduce provide a faster and more efficient way of accomplishing this.

The map() function iterates over all elements in an iterable (list, tuple, set, string) and applies a function to each, and returns a new iterator of the new elements. You can then work with the iterator in the usual way or construct a new list or set from the iterator. Typically, lambda functions are used to specify the function to apply in map.

The filter() function iterates over the elements an iterable (list, tuple, set, string) and applies the fn() function to each element. It returns an iterator for the elements where the fn() returns True.

Comparing list comprehensions vs map / filter
https://towardsdatascience.com/comparing-list-comprehensions-vs-built-in-functions-in-python-which-is-better-1e2c9646fafe

1.6 Tuples

File to use: `tuple-basics.py`

A tuple is basically an immutable list, i.e. a list whose contents cannot be changed. The notation for tuples is parentheses () instead of square brackets [] (for lists). We can perform all the standard operations of a list on a tuple, except that any operations that involve modification.

Advantages of tuples over lists:

a) We generally use tuples for heterogeneous (different) data types and lists for homogeneous (similar) data types.
b) Since tuples are immutable, iterating through a tuple is faster than with a list. So there is a slight performance boost.
c) Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.
d) If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

1.7 Packing / unpacking

File to use: `packing-unpacking.py`

Packing / unpacking refers to extracting individual items out from an iterable (list / tuple) or placing multiple items into an iterable. This can be done using standard loop operations but there is a special packing / unpacking syntax using the * operator to make the code shorter and more precise.

1.8 Variable argument lengths (variadic parameters) with *args

File to use: `variadic-parameters.py`

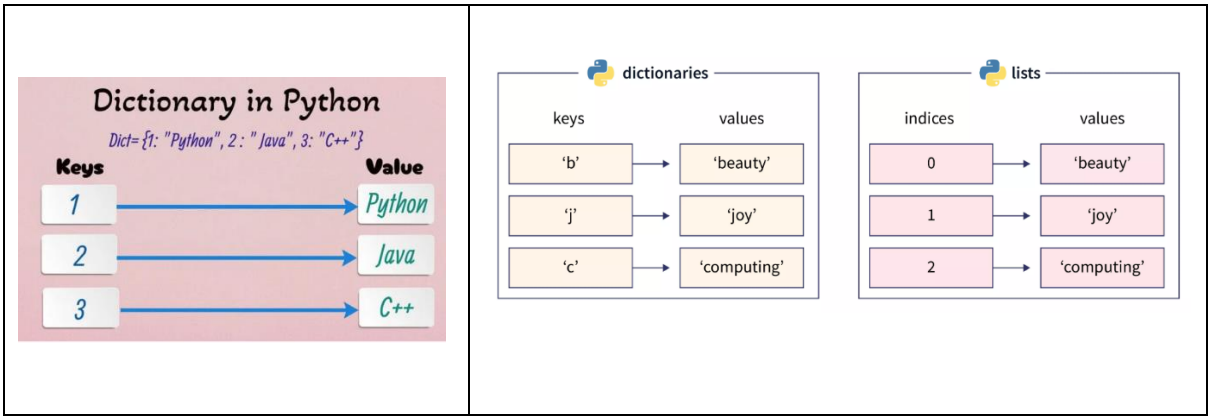
When a function has a parameter preceded by the * operator, it can accept a variable number of arguments through packing / unpacking syntax (as we have just studied earlier). Therefore, you can pass zero, one, or more arguments to the *args parameter

In Python, parameters like *args are called variadic parameters. Functions that have variadic parameters are called variadic functions.

2 Dictionaries

File to use: `dictionary-basics.py`

A dictionary is an unordered collection of key-value pairs where each key is associated with a value. We then use the key to lookup its associated value.
The key in the key-value pair must be an immutable type (for e.g. a number, a string, a tuple, etc). All keys must be unique in a given dictionary - if a key is used more than once, subsequent entries will overwrite the previous one.
The value in the key-value pair can be a number, a string, a list, a tuple, or even another dictionary.



Dictionaries are mutable, so you can add and remove key-value pairs from them anytime, as well as change the values of the key-value pairs.

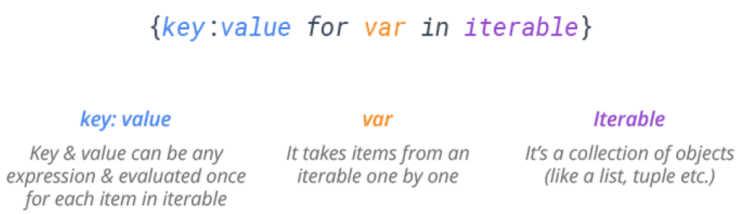
The list of all methods available to a dictionary object are:
https://www.programiz.com/python-programming/methods/dictionary

2.1 Dictionary comprehensions

File to use: `dictionary-comprehension.py`

Dictionary comprehension is an elegant way to define and create lists based on existing dictionaries. It is generally more compact and faster than normal functions and loops for creating dictionaries. This is conceptually identical to list comprehensions

The general syntax for dictionary comprehension looks like this:



Issues with using comprehensions
https://towardsdatascience.com/5-wrong-use-cases-of-python-list-comprehensions-e8455fb75692

2.2 Dictionary as keyword arguments using **kwargs

File to use: `kwargs-parameters.py`

This is based on the concept of packing / unpacking that we saw earlier for lists, but this time using the ** operator, which allows a function to accept a variable number of keyword arguments as a dictionary.

2.3 Sorting a list / dictionary

File to use: `sort-list-dict.py`

Sorting is normally associated with a list, so the result returned from either the `sort` or `sorted` function is another list.
The default sort order is ascending, which can be changed with the `reverse` parameter

String characters are stored internally in memory using a specific encoding system. The traditional encoding system was ASCII
https://www.asciitable.com/
Python's string type uses the modern Unicode Standard based on UTF-8 encoding for representing characters (representing every single character in every single language in the world)
https://home.unicode.org/
The standard alphanumeric characters are still represented using ASCII code in UTF-8 encoding, and sorting is therefore based on this code value.

For a list of complex items (such as other lists or dictionaries), we will need to use a custom sort using the `key` parameter and passing a lambda function to it. The same comment applies to sorting a dictionary.