Python - Notes IV - Data Structures

All data in a computer are represented as strings of binary digits. A **data type** (of a particular programming language) is a built-in way of interpreting such strings of binary digits, that is, it is a classification of the data which carries information about the values it can assume and operations that can be performed on it. Below we will introduce several basic data types built-in to Python: integer, floating point number, Boolean variable and string. Then we will proceed to discuss some more complicated data structures: list, tuple, set, and dictionary which build on the basic data types above.

Integers or floating point numbers.

Integers take values in $\mathbb{Z}$ and **floating point numbers** are given by finite decimal expansions.

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`**`	exponentiation
`%`	modulo
`//`	quotient rounded down to an integer
`abs()`	absolute value

`<`	less than
`>`	greater than
`<=`	less than or equal
`>=`	greater than or equal
`==`	equal
`!=`	not equal

Boolean variable.

Boolean variables can only take two values: `True` or `False`.

`and`	both arguments `True` for output to be `True`
`or`	either argument `True` for output to be `True`
`not`	output is `False` if input is `True` and vice versa

Strings.

Strings are sequential collections of zero or more letters, numbers and other symbols. We call these letters, numbers and other symbols **characters**. Strings are **immutable**, meaning that they cannot be modified through indexing and assignment. Indices for strings start counting at `0`.

EXAMPLE: `"transmogrify"`

`[ ]`	Access an element of a sequence
`+`	Concatenate sequences together
`*`	Concatenate a sequence a repeated number of times
`in`	Ask whether a subsequence is in a sequence
`len`	Ask the number of characters in the sequence
`[ : ]`	Extract a part of a sequence, start(inclusive):stop(exclusive)
`[ : : ]`	Extract a part of a sequence, start(inclusive):stop(exclusive):step

`stringname.center(w)`	Returns a string centered in a field of size `w`
`stringname.count(substring)`	Returns the number of occurrences of `substring`
`stringname.ljust(w)`	Returns a string left-justified in a field of size `w`
`stringname.join(list)`	Concatenates `list` with `stringname` as a separator
`stringname.lower()`	Returns a string in all lowercase
`stringname.upper()`	Returns a string in all uppercase
`stringname.rjust(w)`	Returns a string right-justified in a field of size `w`
`stringname.find(substring)`	Returns the index of the first occurrence of `substring`
`stringname.split(character)`	Splits a string into substrings at `character`
`stringname.replace(string1,string2)`	Replaces all instances of `string1` with `string2`
`list(stringname)`	Converts `stringname` into a list
`sorted(stringname)`	Converts `stringname` into a sorted list
`stringname[::-1]`	Reverses `stringname`
`"".join(reversed(stringname))`	Reverses `stringname`

Lists.

A **list** is an ordered (that is, sequential) collection of zero or more references to Python data objects (such as those introduced previously). Lists are written as comma-delimited values enclosed in square brackets. Lists are **heterogeneous**, meaning that the data objects need not all be the same type (in fact, the data objects may be other lists). The empty list is represented by `[ ]`. Indices for lists start counting at `0`.

EXAMPLE: `[1,3,True,6.5]`

`[ ]`	Access an element of a sequence
`+`	Concatenate sequences together
`*`	Concatenate a sequence a repeated number of times
`in`	Ask whether a subsequence is in a sequence
`len`	Ask the number of characters in the sequence
`[ : ]`	Extract a part of a sequence, start(inclusive):stop(exclusive)
`[ : : ]`	Extract a part of a sequence, start(inclusive):stop(exclusive):step

`listname.append(item)`	Adds `item` to the end of a list
`listname.insert(i,item)`	Inserts `item` at the ith position in a list
`listname.pop()`	Removes and returns the last item in a list
`listname.pop(i)`	Removes and returns the ith item in a list
`listname.sort()`	Sorts a list
`listname.reverse()`	Sorts a list in reverse order
`del listname[i]`	Deletes the item in the ith position
`listname.index(item)`	Returns the index of the first occurrence of `item`
`listname.count(item)`	Returns the number of occurrences of `item`
`listname.remove(item)`	Removes the first occurrence of `item`

Tuples.

Tuples are very similar to lists in that they are heterogeneous sequences of data. A major difference between lists and tuples is that lists can be modified while tuples cannot. This is referred to as mutability. Lists are mutable, whilst tuples are immutable. For example, you can change an item in a list by using indexing and assignment, with a tuple that change is not allowed. With this exception tuples can use any operation described for lists. Tuples are written as comma-delimited values enclosed in parentheses. Indices for tuples start counting at `0`.

Example: `(1,3,True,6.5)`

Sets.

A **set** is an unordered collection of zero or more immutable Python data objects. Sets do not allow duplicates and are written as comma-delimited values enclosed in curly brackets. Sets are heterogeneous. The empty set is represented by `set()`. We can translate a list to a set as follows: `set(listname)`.

EXAMPLE: `{1,True,7.86,"Admiral"}`

`in`	Set membership
`len`	Returns the cardinality of the set
`setname | anotherset`	Returns a new set with all elements from both sets
`setname & anotherset`	Returns a new set with only those elements common to both sets
`setname - anotherset`	Returns a new set with all items from the first set not in second
`setname <= anotherset`	Asks whether all elements of the first set are in the second

`setname.union(anotherset)`	Returns a new set with all elements from both sets
`setname.intersection(anotherset)`	Returns a new set with only those elements common to both sets
`setname.difference(anotherset)`	Returns a new set with all items from first set not in second
`setname.issubset(anotherset)`	Asks whether all elements of one set are in the other
`setname.add(item)`	Adds `item` to the set
`setname.remove(item)`	Removes `item` from the set
`setname.pop()`	Removes an arbitrary element from the set
`setname.clear()`	Removes all elements from the set

Dictionaries.

Dictionaries are unordered collections of associated pairs of items where each pair consists of a key and a value. This key-value pair is typically written as `key:value`. Dictionaries are written as comma-delimited `key:value` pairs enclosed in curly braces. The empty dictionary is represented as {}.

Example: `{"France":"Paris","Spain":"Madrid","Philippines":"Manila"}`

`dictionaryname[k]`	Returns the value associated with `k`, otherwise it's an error
`key in dictionaryname`	Returns `True` if `key` is in the dictionary, or `False` otherwise
`del dictionaryname[key]`	Removes the corresponding entry from the dictionary
`dictionaryname.keys()`	Returns a sequence of the keys
`dictionaryname.values()`	Returns a sequence of the values
`dictionaryname.items()`	Returns a sequence of the key-value pairs
`dictionaryname.get(k)`	Returns the value associated with `k`, or `None` otherwise
`dictionaryname.get(k,alt)`	Returns the value associated with `k`, or `alt` otherwise