```
my_list = ['comp', '125', 'midterm']
my_string = '##'.join(my_list)
print(my_string)
output: comp##125##midterm
```

String methods:

- str.isupper() Returns True if all the characters in str are uppercase, False otherwise
- str.islower() Returns True if all the characters in str are lowercase, False otherwise
- str.isalpha() Returns True if all the characters in str are letters, False otherwise
- str.isdigit() Returns True if all the characters in str are digits False otherwise
- str.upper() str.lower() Returns str with all letters converted to uppercase and lowercase, respectively The original string remains unchanged
- str.strip(char) Returns a copy of the string with all instances of char in the beginning and end removed, if the character char is unspecified the default is a space
- str.lstrip(char), str.rstrip(char) Returns a copy of the string with all instances of char in the begining and end of the string respectively, if char is not specified, the default is an empty space (also including tab characters)
- str.startswith(substring) Returns true if the string starts with the specified substring
- str.endswith(substring) Returns true if the string ends with the specified substring
- str.find(substring) Returns the lowest index in the string where substring is found. If substring is not found, it return -1.
- str.rfind(substring) behaves like a reverse find, searches in reverse from the end of the string and returns the 'highest' index where the substring occurs. Returns -1 otherwise.
- str.replace(old, new) Returns a copy of the string where all instances of old replaced by new.

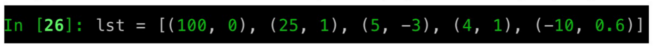
List Methods:

- lst.append(item) Adds item to the end of the list.
- lst.index(item) Returns the index of the first element whose value is equal to item. A ValueError exception is raised if item is not found in the list
- lst.insert(index, item) Insert items into the list at the specified index. When an item is inserted into a list, the list is expanded in size to accommodate the new item. The item that was previously at the specified index, and all the items after it, are shifted by one position toward the end of the list. No exceptions will occur if you specify an invalid index. If you specify an index beyond the end of the list, the item will be added to the end of the list. If you use a negative index that specifies an invalid position, the item will be inserted at the beginning of the list.
- lst.sort(key=, reversed=False/True) sorts the items in the list so they appear in ascending order (from the lowest value to the highest value). An optional key to sort the list based on can be specified using lambda. An optiona method is reversed=True which will reverse the order of sorting.
- lst.remove(item) Removes the first occurrence of item from the list. A valueError exception is raised if item is not found in the list.
- lst.reverse(item) Reverses the order of the items in the list.

Dictionary Methods:

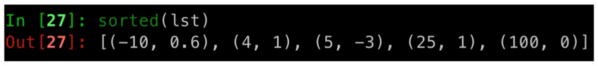
- d.keys(): Returns an iterable that holds all keys
- d.values(): Returns an iterable that holds all values
- d.items(): Returns an iterable that holds all key-value pairs as tuples
- len(d): Returns the number of key-value pairs inside the dictionary
- del statement: Removes a key-value pair (example: del d[key])

Say you have a list of coordinate points (represented by tuples):



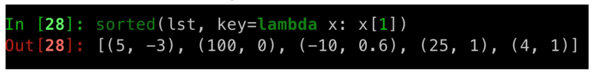
```
In [26]: lst = [(100, 0), (25, 1), (5, -3), (4, 1), (-10, 0.6)]
```

Sort them based on the x-coordinate:



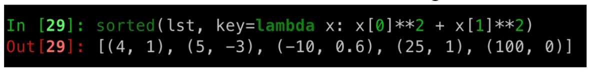
```
In [27]: sorted(lst)
Out[27]: [(-10, 0.6), (4, 1), (5, -3), (25, 1), (100, 0)]
```

Sort them based on the y-coordinate:



```
In [28]: sorted(lst, key=lambda x: x[1])
Out[28]: [(5, -3), (100, 0), (-10, 0.6), (25, 1), (4, 1)]
```

Sort them based on their distance from the origin:



```
In [29]: sorted(lst, key=lambda x: x[0]**2 + x[1]**2)
Out[29]: [(4, 1), (5, -3), (-10, 0.6), (25, 1), (100, 0)]
```

Sorting a list of tuples: what about sorting by 2nd element?

```
lst = [('mango', 3), ('apple', 6), ('lychee', 1), ('apricot', 10)]
res = sorted(lst, key= lambda x: x[1])
[('lychee', 1), ('mango', 3), ('apple', 6), ('apricot', 10)]
```

lst.sort(key= lambda x: x[1]) also does the same thing, but modifies lst permanently

- d.clear(): Removes all the elements from the dictionary
- d.copy(): Returns a copy of the dictionary
- d.get(key): Returns the value of the specified key
- d.pop(key): Removes the element with the specified key
- d.popitem(): Removes the last inserted key-value pair
- dict.fromkeys(keys, values): Returns a dictionary with the specified keys and values
- d.update(pair_iterable): Updates the dictionary with the specified key-value pairs

File IO:

- with open(file_name, mode) as f: ... (automatically closes the file)
- f = open(file_name, mode) ... (must use f.close() after you are done with the file)
- If you open f in read mode, you can use a for loop (example: for line in f: where f was the variable you used) and use f.readline() to read the lines one by one
- If you open f in write mode, use f.write(string). Note that write admits only one parameter and does not start on a new line everytime it is called (unlike print) hence in most cases, you will need to specify '\n' explicitly

OOP Pointers:

Class variables and their values are shared between all instances of a class, example:

```
class Rabbit(Animal):
        tag = 1
        def __init__(self, age, parent1=None, parent2=None):
        super().__init__(age) #Safely use super parent class initialization
        ... (your remaining class initializations)
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

Here tag is used to give a unique id to each new rabbit instance and tag is a class variable while self.rid (which uses tag) is an instance variable.
**note: incrementing class variable changes it for all instances (last line)

Complexity of Common Python Functions

Lists: n is len(L)

- index O(1)
- length O(1)
- append O(1)

- == equal. comparison O(n)
- remove O(n)
- copy O(n)
- reverse O(n)
- iteration O(n)
- in list O(n)

Dictionaries: n is len(d)

- Worst case
  - index O(n)
  - store O(n)
  - length O(1)
  - delete O(n)
  - iteration O(n)
- Average case
  - index O(1)
  - store O(1)
  - delete O(1)
  - iteration O(n)

Assertions and Exception handling:

```
try:

        statements_that_may_cause_an_exception

except ValueError as err:

        use_the_error_information_as_you_wish

except:

        statements_to_handle_the_exception

else:

code_to_run_when_no_exceptions_occur

finally:

        code_to_run_always_regardless_of_an_error_occured_or_not
```

```
fltval = 123.45789
strval = 'programming'

print(f'Truncate to two digits after decimal in {fltval :.2f}')
print(f'Can even do some operation and then format result {fltval*fltval :.3f}')
print(f'Shortened strings are {strval :.5} and {strval :.8}')
print(f'Cannot make it longer: {strval :.25}')
```

Output:

```
Truncate to two digits after decimal in 123.46
Can even do some operation and then format result 15241.851
Shortened strings are progr and programm
Cannot make it longer: programming
```