Comprehensions:

* A way to conveniently create one list from another (also works with sets and dictionaries)
* list1 = [1,2,3,4,5]
* list2 = [i\*2 for i in list1]
* dict1 = {i : i\*\*3 for I in list1}
* set1 = {i\*10 for I in list1}
* Tuple Comprehensions don’t exist, instead we have Generator Comprehensions.
  + gen1 = (i + 5 for i in list1)
  + Generators are more memory efficient because they don’t allocate memory for the whole collection but instead generate each element individually.
* Nested Comprehensions:
  + A way to make comprehensions using nested lists:
  + Ex.
    - matrix = [[j for j in range(5)] for i in range(5)]
    - Output:
      * print(matrix)
      * [[0,1,2,3,4],[0,1,2,3,4],[0,1,2,3,4],[0,1,2,3,4],[0,1,2,3,4]]

Lambdas:

* A small, anonymous function.
* They can have any number of inputs but must have only one expression, which is the returned object.
* Ex.
  + lambda a,b: a\*b
* Lambdas can be used in any situation where a function is required, such as the input of a filter() or map() function.

Closures:

* A function object that remembers values in enclosing scopes even if they are not present in memory.

# Python program to illustrate

# closures

**def** outerFunction(text):

**def** innerFunction():

**print**(text)

    # Note we are returning function

    # WITHOUT parenthesis

**return** innerFunction

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    myFunction **=** outerFunction('Hey!')

    myFunction()

* Notice how the return from outerFunction() is not a value, but a function object.
* This allows functions to be invoked outside of their scope.
  + By returning innerFunction, outerFunction’s scope is extended to allow for the use of any input given to innerFunction.

**import** logging

logging.basicConfig(filename**=**'example.log',

                    level**=**logging.INFO)

**def** logger(func):

**def** log\_func(**\***args):

        logging.info(

            'Running "{}" with arguments {}'.format(func.\_\_name\_\_,

                                                    args))

**print**(func(**\***args))

    # Necessary for closure to

    # work (returning WITHOUT parenthesis)

**return** log\_func

**def** add(x, y):

**return** x**+**y

**def** sub(x, y):

**return** x**-**y

add\_logger **=** logger(add)

sub\_logger **=** logger(sub)

add\_logger(3, 3)

add\_logger(4, 5)

sub\_logger(10, 5)

sub\_logger(20, 10)

* Why use closures:
  + They provide some sort of data hiding, which reduces the need for global variables.
  + They make code very efficient if it has a small number of functions.

I/O Operations:

* Python has many built-in methods for I/O operations.
* Console I/O:
  + print()
  + input()
  + raw\_input()
* File handling:
  + open(path, ‘mode’)
    - Can open in different ‘modes’:
      * ‘r’ – read only.
      * ‘r+’ – read and write.
      * ‘w’ – write only. For existing files, the data is overwritten. Creates non-existent files.
      * ‘w+’ – read and write. For existing files, the data is overwritten.
      * ‘a’ – append only. For existing files, the new data is inserted at the end of the file. Creates non-existent files.
      * ‘a+’ – append and read. For existing files, new data is inserted at the end of the file. Creates non-existent files.
      * ‘x’ – exclusive creation. Creates and writes to a file. Will cause an error if the file already exists.
      * ‘x+’ – same as ‘x’ but it is now readable as well.
      * Add a ‘b’ to any of these modes to perform that operation using Binary format.
  + write()
  + writelines()
  + read()
  + readline()
  + readlines()
  + tell()
    - returns the current position of the cursor.
  + seek(offset, from)
    - changes the cursor position.
    - Offset means how many bytes to move.
    - From means where the cursor starts:
      * 0 = start of file
      * 1 = current location
      * 2 = end of file
  + close()
* Helpful methods from OS module:
  + os.rename(old\_file\_name, new\_file\_name)
  + os.remove(file\_name)
  + os.mkdir(dir\_name)
  + os.chdir(dir\_name)
  + os.getcwd()
  + os.rmdir(dir\_name)
* Streams:
  + Python has 3 main types of I/O: text, binary, and raw.
  + Text:
    - Expects and produces string objects.
  + Binary:
    - Expects and produces bytes like objects.
  + Raw:
    - Typically used as low-level building blocks for text/binary streams.
  + Set the binary stream buffering key argument to 0 and a text stream will automatically add “\r\n” to the end of a line (binary will not).