```
import timeit   # create new namespace as a container for all obj; execute the code;
import array, math, fractions, random
import timeit as timer  # custom name to refer to a module
from timeit import foo  # load specific definition within a module
from timeit import *    # load all definitions except those that start with an underscore

'''COMPARISON'''
w < x < y < z   # the same as
w < x and x < y and y < z
x == y  # equal value
x is y    # equal obj in memory
z = x if x < y else y      # conditional expression

'''NAMESPACE'''
all_global = globals().keys()      # get all global objects as one dictionary
all_global_one = globals()['string_one']  # get object from global scope
globalVarNameOne = 4
GlobalVarNameTwo = 8
def funcName():
    global globalVarNameOne
    globalVarNameOne = 14       # modify global variable
    GlobalVarNameTwo = 0        # create a new local variable

def countDown(start):
    n = start
    def decrement():      # nonlocal does not bind a name to a local variable
        nonlocal n
        n -= 1

'''OPERATORS WITH ANY SEQUENCE TYPES'''
objOne + objTwo          # concatenation
objOne * n        # makes n copies of objOne
varOne, varTwo, varThree = objOne      # variable unpacking
varOne in objOne         # membership
for x in objOne:         # iteration
all(objOne)       # return True if all items are true
any(objOne)       # return True if any item is true
len(objOne)       # length
min(objOne)       # min/max value in objOne
sum(objOne [,initial])    # summ of items with an optional initial value

'''STRING IMMUTABLE OBJECTS'''
string_one = "Python version: {x}.{y}".format(x=3, y=14)
string_two = "Python version: %s" % time.ctime()
stringName.capitalize()            # capitalizes the first character
stringName.count(sub [,start [,end]])    # counts occurences of the specified substring
stringName.startswith(prefix [,start [,end]])       # checks whether a string starts with prefix
stringName.endswith(suffix [,start [,end]])         # checks the end of the string for a suffix
stringName.find(sub [,start [,end]])       # finds the first occurrence of the specified sub
stringName.rfind(sub [,start [,end]])      # finds the last occurrence of a sub
stringName.split([sep [,maxsplit]])        # splits a str using separator as a delimiter
stringName.rsplit(sep, [,maxsplit])        # splits a str from the end using separator
stringName.isalnum()    # whether all chars are alphanumeric
stringName.isalpha()    # whether all chars are alphabetic
stringName.isdigit()      # whether all chars are digits
stringName.join(separator)       # joins the string with a separator
stringName.lower()       # to lower case, to upper case
stringName.replace(oldSub, newSub [,maxreplace])      # replace a substring

'''LIST MUTABLE OBJECT'''         # can hold any data type in one list, for c-lie array use array module
del listName[index]      # deletes an element
del listName[indexStart : indexEnd]      # deletes a slice
listName.append(obj)               # add one object to the end
listName.extend(newListName)            # add a new list to the end
listName.count(obj)       # counts occurrences of obj
listName.index(obj [,start [,stop]])        # returns the smallest index of obj
listName.insert(index, obj)        # inserts obj at index
listName.pop([index])              # return elem on index and remove it from the list
listName.remove(obj)     # remove obj
listName.reverse()        # reverses items in place
listName.sort([sortFunc [,reverse]])       # sort list of items
```

```
'''DICTIONARY OBJECT'''          # Key values can be any immutable object (string, number, tuple)
dict_all_keys = dictName.keys()          # make a list of all keys/values/items
del dictName[key]        # removes from dict
key in dictName          # returns true if key is here
dictName.clear()         # removes all items
dictName.copy()          # copy
dictName.fromkeys(sequence [,value])          # create new dict with keys from sequence and values all set to value
dictName.get(key [,value])       # returns dictName[key], otherwise value
dictName.pop(key [,default])             # returns dictName[key] and removes it from dictName
dictName.update(dictNewName)             # add all obj from dictNewName to dictName

'''SET OBJECT'''
setName = {1, 2, 3}      # unordered collection of unique items
setName.copy()           # copy
setName.difference(setNewName)       # returns all the items in setName but not in setNewName
setName.intersection(setNewName)     # returns all the items that are both in two sets
setName.isdisjoint(setNewName)       # returns true if both have no items in common
setName.issubset(setNewName)         # true if setName is a subset of setNewName
setName.isuperset(setNewName)        # true if setName is a superset of setNewName
setName.symmetric_difference(setNewName)             # return all items that are in first or second set but not in both
setName.union(setNewName)            # return all items in setNewName or setName

'''TUPLE IMMUTABLE OBJECTS'''
tuple_one = (1, 1, 2, 3)             # immutable ordered collection
tuple_count = tuple_one.count(1)         # calc how many times arg take place in tuple

'''SEQUENCE ITERATOR'''
matrix_one = [[1, 1, 1], [4, 5, 6], [7, 8, 9]]
matrix_two = [(1, 2), (3, 4), (5, 6)]

l_comp_one = [row[0] for row in matrix_one]              # take the first element in each row
l_comp_two = [(letter+'a') for letter in 'Hi']               # what you want + how you call it + where
l_comp_three = [(z+1) for z in range(5) if z % 2 == 0]            # what you want + how you call it + where + if
l_comp_four = [(x,y) for x in listA for y in listB]

g_express_one = (10 * i for i in matrix_one)      # generator expression
g_express_one.next()

while expression:
    pass
for x in range(10):      # generator function (will not save values in ram)
    pass
for y in matrix_one:     # work with any sequence
    pass
for (a, b) in matrix_two:            # tuple unpacking
    pass
for (key, value) in dict_one.items():      # dictionary iteration
    pass

for index, value in enumerate(matrix_one):      # iterator that returns sequence of tuples (index, value)
    matrix_one[index] = value * value

for x, y in zip(listOne, listTwo):               # combines two lists into a sequence of tuples
    # (listOne[0], listTwo[0]), (listOne[1], listTwo[1])

for x in matrix_one:     # else will be executed if loop is runs to completion
    if not True:
        break # else clause is skipped
else:
    raise RuntimeError("Error")

'''DECORATORS'''
def decorator_func(func):
    def insider_func():              # add logic before and after decorated func call
        print('Code here, before executing the func')
        func()
        print('Code here will execute after all')
    return insider_func

@decorator_func #more than one can be applied
def any_func_name():
    print('This function needs a decorator')
```

```
'''CLASSES AND OBJECTS'''       # functions (methods), variables (class variables), computed attributes (properties)
class SampleClass(object):
    species = 'Human'               # class object attribute the same for all instances

    @staticmethod
    def static_method(arg): pass

    @classmethod
    def class_method(cls, arg): pass

    def __init__(self, name, last_name):
        self.__name = name
        self.__last_name = last_name

    @property    # property getter, accessed as instanceName.name
    def name(self):
        return self.__name

    @name.setter          # property setter, accessed as instanceName.name = "new str"
    def name(self, value):
        if not isinstance(value, str):
            raise TypeError("Must be a string!")
        self.__name = value

    @name.deleter
    def name(self):
        raise TypeError("Cannot delete name")

    def instance_method(self):
        pass

# create a callable obj that wraps both a method and an associated instance
classInstance = SampleClass("Sergey", "Melentyev")
boundMethod = classInstance.instance_method
boundMethod()

# create a callable obj that wraps the method, but expecs an instance of the propper type to be passed
unboundMethod = SampleClass.instance_method
unboundMethod(classInstance, "Sergey", "Melentyev")

class Sample_Sub_Class(SampleClass):
    def __init__(self, name, last_name, second_name):    # sub class constructor
        SampleClass.__init__(self, name, last_name)          # call super class constructor
        self.second_name = second_name

    def sub_instance_method(self):
        super().instance_method()            # call super class method
        pass

'''CONTEXT MANAGER AND WITH'''       # items = [1, 2, 3]
class ListTransaction(object):
    def __init__(self, theList):
        self.theList = theList
    def __enter__(self):
        self.workingCopy = list(self.theList)
        return self.workingCopy
    def __exit__(self, type, value, tb):
        if type is None:
            self.theList[:] = self.workingCopy
        return False

with ListTransaction(items) as working:          # will produce [1, 2, 3, 4, 5]
    working.append(4)
    working.append(5)

try:
    with ListTransaction(items) as working:       # will produce [1, 2, 3, 4, 5]
        working.append(6)
        working.append(7)
        raise RuntimeError("Something happened!")
except RuntimeError:
    pass
```

```
'''EXCEPTION HANDLING'''
try:
    answer = 2 + 'a'
except TypeError:       # check full list of build-in exceptions
    print("This will be printed in case of TypeError acquire")
else:
    print("This will be printed if no errors acquire")
finally:
    print("This will be printed in any case")

try:
    # catch all exceptions in one place
except Exception as e:
    print("An error: {err}\n".format(err=e))

class MyOwnErrorType(Exception):            # create a custom exception
    def __init__(self, errno, msg):
        self.args = (errno, msg)
        self.errno = errno
        self.errmsg = msg

class HostNameError(MyOwnErrorType): pass
class TimeOutError(MyOwnErrorType): pass
def errorOne(): raise HostNameError("Unknown host")
def errorTwo(): raise TimeOutError("Timed out")
try:
    errorOne()
except MyOwnErrorType as e:
    if type(e) is HostNameError:
        #logic here

class ListTransaction(object):
    def __init__(self, theList):
        self.theList = theList
    def __enter__(self):
        self.workingCopy = list(self.theList)
        return self.workingCopy
    def __exit__(self, type, value, tb):
        if type is None:
            self.theList[:] = self.workingCopy
        return False

'''BUILD-IN FUNCTIONS'''
# map() apply a func to every item in a list, return a list of all items
lambda_function = lambda arg_one,arg_two: arg_one + arg_two
sample_list = [0, 22.5, 40, 100]
mapped_lambda = list(map(lambda arg: (9.0/5*arg + 32), sample_list))

def sample_function(arg): return (9.0/5)*arg + 32
mapped_list = list(map(sample_function, sample_list))

# reduce() apply a func to every item in a list in pare of two, return only one final item
# filter() apply a func that return a bool to the list in pare of two, return only one final item
# zip() combine items at each index in a tuple from two lists
# all() return True if all elements are true
# any() return True if any element is true

'''GENERATORS'''
# yield = return in order to keep track only on current call


# check speed of a function
timer = timeit.timeit("'-'.join(str(n) for n in range(100))", number=1000)
```