Collections

```
      #     Collections are data structures
      #     A collection represents a number of individual values

      #     Simple examples of collections are sequences like: strings, lists and tuples
C001 >>>    string = 'Hello World'
C002 >>>    string
     ==>    'Hello World'
C003 >>>    list(string)  # conversion into a list of single letters
     ==>    ['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']

C004 >>>    mylist = ['text', 77, False, 23.7777]  # lists may contain different data types
C005 >>>    for item in mylist:    # lists (all sequences) support the for ... in ... syntax
     ...        print("item:", item, '  type:', type(item))
     p()    item: text   type: <class 'str'>
     p()    item: 77   type: <class 'int'>
     p()    item: False   type: <class 'bool'>
     p()    item: 23.7777   type: <class 'float'>
C006 >>>    77 in mylist    # sequences support the 'in' syntax
     ==>    True
C007 >>>    'sense' in mylist  # 'a in b' is a boolean expression, can be True or False
     ==>    False
C008 >>>    False in mylist
     ==>    True
C009 >>>    True in mylist
     ==>    False
```

Sequence Types: Tuples and Lists

```
      #     Sequences allow to handle a number of values with a single name (=variable)
      #     This makes it different from a skalar, which is one single value
      #     Sequences support a number of different operations. Some examples for this:

      #     The logical 'in' operation returns a boolean value (True or False)
C010 >>>    capitals = ['rome', 'london', 'paris', 'berlin', 'lisbon', 'madrid']
C011 >>>    'porto' in capitals
     ==>    False
C012 >>>    'paris' in capitals
     ==>    True
C013 >>>    primes = (2,3,5,7,11,13,17,19,23,29,31,37,41,43,47)
C014 >>>    for n in range(10):
     ...        if n in primes:
     ...            print("{} is a prime number".format(n))
     p()    2 is a prime number
     p()    3 is a prime number
     p()    5 is a prime number
     p()    7 is a prime number
C015 >>>    valid_list = [1, 1, 1, 7, 7, 1, 1, 3, 4, 5, 7]  # equal elements allowed

      #     A string can be considered a sequence as well. The 'in' operation works:
C016 >>>    'w' in 'portwine'
     ==>    True
C017 >>>    'twin' in 'portwine'
     ==>    True
```

Sequence Types: Indexing and Slicing

```
      #     Sequences allow to access (read) single or a group of elements
C018 >>>    capitals = ['rome', 'london', 'paris', 'berlin', 'lisbon', 'madrid']
C019 >>>    capitals[2]    # the third element
     ==>    'paris'
C020 >>>    capitals[-1]    # the last element
     ==>    'madrid'
C021 >>>    primes = (2,3,5,7,11,13,17,19,23,29,31,37,41,43,47)
C022 >>>    primes[2:4]  # includes the third, but not the fifth
     ==>    (5, 7)
C023 >>>    primes[:3]    # all elements up to, but not including the fourth
     ==>    (2, 3, 5)
C024 >>>    primes[10:]  # the eleventh and all following
     ==>    (31, 37, 41, 43, 47)
      #     the zero-based counting (the first is 0) may be mind-twisting, but only in the beginning :-)

C025 >>>    capitals.index('berlin')
     ==>    3
C026 >>>    capitals.index('milano')
     err!   ValueError("'milano' is not in list",)
C027 >>>    min(capitals), max(capitals)  # this only works, when all elements are 'comparable'
     ==>    ('berlin', 'rome')
C028 >>>    'popocatepetl'.count('p')    # another operation on sequences
     ==>    3
```

Sequence Types: Tuples (1)

```
      #     A tuple is an immutable sequence, it can not be modified
      #     A tuple is constructed in several ways:
C029 >>>    primes = (2,3,5,7,11,13,17,19)   # a number of values enclosed in '(...)'
C030 >>>    astring = 'abcde'
C031 >>>    tuple(astring)    # the tuple function takes the argument as a sequence
     ==>    ('a', 'b', 'c', 'd', 'e')
C032 >>>    tuple()  # create an empty tuple
     ==>    ()
C033 >>>    ()    # another way to create an empty tuple
     ==>    ()
C034 >>>    (astring, astring)  # this creates a tuple with 2 elements
     ==>    ('abcde', 'abcde')
C035 >>>    (astring)    # this is not a tuple
     ==>    'abcde'
      #     use a comma to distinguish a one-element-tuple from an expression in parenthesis
C036 >>>    (astring,)
     ==>    ('abcde',)
```

Sequence Types: Tuples (2)

```
      #     Tuples can be created and unpacked by lists of elements:
C037 >>>    p1, p2, p3, p4, p5, p6, p7, p8 = primes  # unpacking
C038 >>>    somep = p1, p4, p7   # packing
C039 >>>    somep
     ==>    (2, 7, 17)
C040 >>>    a = 10; b = 20
C041 >>>    a, b = b, a    # packing can be used to exchange values
C042 >>>    "a:{}, b:{}".format(a, b)
     ==>    'a:20, b:10'

C043 >>>    primes[3] = 22   # immutable
     err!   TypeError("'tuple' object does not support item assignment",)
C044 >>>    astring[4] = 'z'   # immutable
     err!   TypeError("'str' object does not support item assignment",)
```

Sequence Types: Lists

```
      #     A list is a mutable sequence of values.
      #     A list is displayed as comma separated values in brackets '[]'
C045 >>>    alist = [1, 4, 7, 8]
C046 >>>    type(alist), alist
     ==>    (<class 'list'>, [1, 4, 7, 8])
      #     lists support indexing, slicing and the 'in' operator
      #     list can be modified
C047 >>>    del alist[0]; print(alist)    # delete on element
     p()    [4, 7, 8]
C048 >>>    alist[0] = 3; print(alist)    # replace one element
     p()    [3, 7, 8]
C049 >>>    alist.append(2); print(alist)  # append one element
     p()    [3, 7, 8, 2]
C050 >>>    alist[3:3] = [6]; print(alist)  # insert one element
     p()    [3, 7, 8, 6, 2]
      #     beware! - these are only basic examples. For more see the python documentation about lists

C051 >>>    sorted(alist)  # this returns a sorted copy of alist
     ==>    [2, 3, 6, 7, 8]
C052 >>>    alist
     ==>    [3, 7, 8, 6, 2]
C053 >>>    alist.sort()  # this sorts the elements of alist itself
C054 >>>    alist
     ==>    [2, 3, 6, 7, 8]
```

Mapping Types: Dictionaries (1)

```
      #     A dictionary is a collection, where each element has a key
C055 >>>    days = {0: 'Monday', 1: 'Tuesday', 2: 'Wednesday', 3: 'Thursday',
     ...            4: 'Friday', 5: 'Saturday', 6: 'Sunday'}
C056 >>>    days[2]   # access one element by its key
     ==>    'Wednesday'
C057 >>>    days[5]
     ==>    'Saturday'
C058 >>>    days[5] = 'Sabado'   # values can be replaced
C059 >>>    days[7] = 'Doomsday'    # new values can be added
C060 >>>    for day in range(8):
     ...        print("day[{}] = '{}'".format(day, days[day]))
     p()    day[0] = 'Monday'
     p()    day[1] = 'Tuesday'
     p()    day[2] = 'Wednesday'
     p()    day[3] = 'Thursday'
     p()    day[4] = 'Friday'
     p()    day[5] = 'Sabado'
     p()    day[6] = 'Sunday'
     p()    day[7] = 'Doomsday'

      #     keys for a dictionary can be strings (and all other static types)
C061 >>>    valid_bets = {'r': 'Rock', 'p': 'Paper', 's': 'Scissor'}
C062 >>>    user_bet = 'p'
C063 >>>    print("Your bet was '{}'".format(valid_bets[user_bet]))
     p()    Your bet was 'Paper'
```

Mapping Types: Dictionaries (2)

```
C064 >>>    empty = {}    # or
C065 >>>    empty = dict()    # which is a bultin function
C066 >>>    newdict = dict(r='Rock', p='Paper', s='Scissor')  # another way to create a dictionary
C067 >>>    newdict  # this only works for keys, that would be valid variable names
     ==>    {'s': 'Scissor', 'p': 'Paper', 'r': 'Rock'}
C068 >>>    newdict.keys()  # dict keys are always 'unordered'
     ==>    dict_keys(['s', 'p', 'r'])
C069 >>>    keylist = sorted(newdict.keys())    # return a sorted copy of the key list
C070 >>>    keylist    # this is, what we get, now use it
     ==>    ['p', 'r', 's']
C071 >>>    for short in keylist:    # print the content of a dictionary
     ...        print("key: {}, value:'{}'".format(short, newdict[short]))
     p()    key: p, value:'Paper'
     p()    key: r, value:'Rock'
     p()    key: s, value:'Scissor'
```

Mapping Types: Dictionaries (3)

```
C072 >>>    newdict = dict(r='Rock', p='Paper', s='Scissor')
       #    like the keys() method, dictionaries also have the values() and the items() methods
C073 >>>    list(newdict.values())
     ==>    ['Scissor', 'Paper', 'Rock']
C074 >>>    list(newdict.items())  # the items() returns tuples of keys + values
     ==>    [('s', 'Scissor'), ('p', 'Paper'), ('r', 'Rock')]
C075 >>>    del newdict['s']  # remove a value from a dictionary
C076 >>>    newdict
     ==>    {'p': 'Paper', 'r': 'Rock'}
```

Set - a collection of unique values

```
      #     sets can be seen as dicitionaries without values, only keys
      #     sets are best explained by examples

C077 >>>    set((0,3,4,5,2,3,6,7,1,2,3,4,5,6,4,3,1))   # each number occurs only once
     ==>    {0, 1, 2, 3, 4, 5, 6, 7}
C078 >>>    set(('abc', 'abd', 'cab','abd', 'acb', 'bca', 'cab')) # a sequence of strings
     ==>    {'abc', 'bca', 'abd', 'acb', 'cab'}
C079 >>>    set('popocatepetl')  # a single string is considered as sequence of characters
     ==>    {'t', 'c', 'o', 'p', 'e', 'a', 'l'}
      #     see: the output of a set is unordered and appears in curly braces - like dictionaries

      #     sets support the 'in' operator and can be iterated
C080 >>>    s = set('popocatepetl')
C081 >>>    len(s)    # number of elements
     ==>    7
C082 >>>    't' in s  # 'in' operator
     ==>    True
C083 >>>    sorted(s)  # sorted() requires an iterable
     ==>    ['a', 'c', 'e', 'l', 'o', 'p', 't']
```

Sets - more of it

```
      #     sets allow for set operations, like union, subset, difference, and more
C084 >>>    hv = set('heaven')
C085 >>>    hl = set('hell')
C086 >>>    hv & hl  # returns a new set, which is an intersection (elements in hv and hl)
     ==>    {'h', 'e'}
C087 >>>    hv | hl  # this is the union (elements in hv or hl or both)
     ==>    {'n', 'v', 'e', 'h', 'a', 'l'}
C088 >>>    hv ^ hl  # symmetric difference (exclusive or, either in hv or in hl)
     ==>    {'n', 'v', 'a', 'l'}
C089 >>>    hv - hl  # a new set with all of hl removed from hv
     ==>    {'n', 'a', 'v'}

      #     what is this?
C090 >>>    pri = set(range(2,1000000))
C091 >>>    for x in range(2,1000):
     ...        if x in pri:
     ...            pri = pri - set(range(x+x,1000000,x))
C092 >>>    len(pri)
     ==>    78498
C093 >>>    for x in range(0, 48, 12): print(sorted(pri)[x:x+12])
     p()    [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
     p()    [41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89]
     p()    [97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151]
     p()    [157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223]
```

Something really 'pythonic': Comprehensions

```
      #   Lists and dictionaries are fundamental in Python. They are virtually everywhere.
      #   That is why Python offers some special syntax to work with lists and dictionaries.

      #   List comprehensions are a way to transform lists. Some examples:
C094 >>>  names = ['tom', 'mike', 'jane', 'trevor', 'susan']

C095 >>>  [n.capitalize() for n in names]  # create a new list, names is not modified
     ==>  ['Tom', 'Mike', 'Jane', 'Trevor', 'Susan']
C096 >>>  [n for n in names if len(n) != 4] # conditions work like filters
     ==>  ['tom', 'trevor', 'susan']
C097 >>>  ['*'*len(n) for n in names]  # the expression left of the 'for' should return a value
     ==>  ['***', '****', '****', '******', '*****']
C098 >>>  [print(n) for n in names if n[0] == 't']  # else, None values are generated
     p()  tom
     p()  trevor
     ==>  [None, None]
```

List comprehensions

```
      #     multiple 'for's create permutations
C099 >>>    perm = [x+y+z for x in 'abc' for y in 'abc' for z in 'abc']
C100 >>>    print('{}\n{}\n{}'.format(perm[:9], perm[9:18], perm[18:]))
     p()    ['aaa', 'aab', 'aac', 'aba', 'abb', 'abc', 'aca', 'acb', 'acc']
     p()    ['baa', 'bab', 'bac', 'bba', 'bbb', 'bbc', 'bca', 'bcb', 'bcc']
     p()    ['caa', 'cab', 'cac', 'cba', 'cbb', 'cbc', 'cca', 'ccb', 'ccc']

C101 >>>    [0 for x in range(10)]   # the left can be a constant, the right side may be any iterable
     ==>    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
C102 >>>    [[0,0,0] for x in range(5)]  # this creates a list of lists
     ==>    [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]

      #     list of lists? - sounds like matrix ...
```

More on list comprehensions

```
      #     its possible to create a matrix
C103 >>>    dimx, dimy = 9, 7
C104 >>>    mat = [[max(x,y) for x in range(dimx)] for y in range(dimy)]
C105 >>>    for row in mat:
      ...       print(row)
      p()   [0, 1, 2, 3, 4, 5, 6, 7, 8]
      p()   [1, 1, 2, 3, 4, 5, 6, 7, 8]
      p()   [2, 2, 2, 3, 4, 5, 6, 7, 8]
      p()   [3, 3, 3, 3, 4, 5, 6, 7, 8]
      p()   [4, 4, 4, 4, 4, 5, 6, 7, 8]
      p()   [5, 5, 5, 5, 5, 5, 6, 7, 8]
      p()   [6, 6, 6, 6, 6, 6, 6, 7, 8]

      #     and we can even transform the complete matrix
C106 >>>    newmat = [[mat[y][x] %3+1 for x in range(len(mat[0]))]
C107 >>>                              for y in range(len(mat))]
C108 >>>    for row in newmat:
      ...       print(row)
      p()   [1, 2, 3, 1, 2, 3, 1, 2, 3]
      p()   [2, 2, 3, 1, 2, 3, 1, 2, 3]
      p()   [3, 3, 3, 1, 2, 3, 1, 2, 3]
      p()   [1, 1, 1, 1, 2, 3, 1, 2, 3]
      p()   [2, 2, 2, 2, 2, 3, 1, 2, 3]
      p()   [3, 3, 3, 3, 3, 3, 1, 2, 3]
      p()   [1, 1, 1, 1, 1, 1, 1, 2, 3]

      #     and list comprehensions are fast!
```

Comprehensions also work for dictionaries

```
C109 >>>    countries = 'belgium portugal turkey greece bulgaria'.split()

       #    let's create a dictionary of counters
C110 >>>    counters = {key: 0 for key in countries}  # see the curly braces
C111 >>>    counters['portugal'] += 10
C112 >>>    for key, value in counters.items():
     ...        print("{:8s} => {}".format(key, value))
     p()    belgium  => 0
     p()    bulgaria => 0
     p()    greece   => 0
     p()    portugal => 10
     p()    turkey   => 0

C113 >>>    from external import get_capitals

       #    we can use the external function to get the capital:
C114 >>>    get_capitals('bulgaria')
     ==>    'sofia'
       #    but let's create a local dictionary
C115 >>>    mydict = {key : get_capitals(key) for key in countries}
C116 >>>    for key, value in mydict.items():
     ...        print("{:8s} => {}".format(key, value))
     p()    belgium  => brussels
     p()    bulgaria => sofia
     p()    greece   => athens
     p()    portugal => lisbon
     p()    turkey   => ankara
```