Overview of Built-In Sequences:

The standard library offers a rich selection of sequence types implemented in C:

*Container sequences:*

list, tuple, and collections.deque can hold items of different types.

*Flat sequences:*

str, bytes, bytearray, memoryview, and array.array hold items of one type.

*Container* *sequences* hold references to the objects they contain, which may be of any type, while *flat* *sequences* physically store the value of each item within its own memory space, and not as distinct objects. Thus, flat sequences are more compact, but they are limited to holding primitive values like characters, bytes, and numbers.

Another way of grouping sequence types is by mutability:

*Mutable sequences*: list, bytearray, array.array, collections.deque, and memoryview

*Immutable sequences*: tuple, str, and bytes

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | Container | | \_\_contains\_\_ | | |  | | --- | | Sequence | | \_\_getitem\_\_ | | \_\_contains\_\_ | | \_\_iter\_\_ | | \_\_reversed\_\_ | | index() | | count() | | |  | | --- | | Mutable Sequence | | \_\_setitem\_\_ | | \_\_delitem\_\_ | | insert() | | append() | | reverse() | | extend() | | pop() | | remove() | | \_\_iadd\_\_ | |
| |  | | --- | | Iterable | | \_\_iter\_\_ | |
| |  | | --- | | Sized | | \_\_len\_\_ | |

List Comprehensions and Generator Expressions:

A quick way to build a sequence is using a list comprehension (if the target is a list) or a generator expression (for all other kinds of sequences). If you are not using these syntactic forms on a daily basis.

Writing code this way is faster and more readable, even if it seems other way.

For brevity, many Python programmers refer to list comprehensions as *listcomps*, and generator expressions as *genexps*. I will use these words as well.

Here is a test: which do you find easier to read, Example 1 or Example 2?

**Example 1: X**

>>> symbols = '$¢£¥€¤'

>>> codes = []

>>> for symbol in symbols:

... codes.append(ord(symbol))

...

>>> codes

[36, 162, 163, 165, 8364, 164]

**Example 2: X**

>>> symbols = '$¢£¥€¤'

>>> codes = [ord(symbol) for symbol in symbols]

>>> codes

[36, 162, 163, 165, 8364, 164]

List Comprehensions and Readability:

A for loop may be used to do lots of different things: scanning a sequence to count or pick items, computing aggregates (sums, averages), or any number of other processing tasks. The code in Example 1 is building up a list. In contrast, a *listcomp* is meant to do one thing only: to build a new list. Of course, it is possible to abuse list comprehensions to write truly incomprehensible code. I’ve seen Python code with listcomps used just to repeat a block of code for its side effects. If you are not doing something with the produced list, you should not use that syntax. Also, try to keep it short. If the list comprehension spans more than two lines, it is probably best to break it apart or rewrite as a plain old for loop. Use your best judgment: for Python as for English, there are no hard-and-fast rules for clear writing

**Syntax Tip:**

In Python code, line breaks are ignored inside pairs of [ ], { }, or ( ). So, you can build multiline lists, listcomps, genexps, dictionaries and the like without using the ugly \ line continuation escape.

Cartesian Products

Listcomps can generate lists from the Cartesian product of two or more iterables. The items that make up the cartesian product are tuples made from items from every input iterable. The resulting list has a length equal to the lengths of the input iterables multiplied.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | [♠, | ♡, | ♢, | ♣,] |
| [A | [A♠, | A♡, | A♢, | A♣, |
| K | K♠, | K♡, | K♢, | K♣, |
| Q] | Q♠, | Q♡, | Q♢, | Q♣] |

>>> colors = ['black', 'white']

>>> sizes = ['S', 'M', 'L']

>>> tshirts = [(color, size) for color in colors for size in sizes]

>>> tshirts

[('black', 'S'), ('black', 'M'), ('black', 'L'), ('white', 'S'),

 ('white', 'M'), ('white', 'L')]

>>> for color in colors:

... for size in sizes:

... print((color, size))

...

('black', 'S')

('black', 'M')

('black', 'L')

('white', 'S')

('white', 'M')

('white', 'L')

>>> tshirts = [(color, size) for size in sizes

... for color in colors]

>>> tshirts

[('black', 'S'), ('white', 'S'), ('black', 'M'), ('white', 'M'),

 ('black', 'L'), ('white', 'L')]

Generator Expressions:

To initialize tuples, arrays, and other types of sequences, you could also start from a listcomp, but a genexp saves memory because it yields items one by one using the iterator protocol instead of building a whole list just to feed another constructor. Genexps use the same syntax as listcomps but are enclosed in parentheses rather than brackets.

**Generator Exp: X**

>>> symbols = '$¢£¥€¤'

>>> tuple(ord(symbol) for symbol in symbols)

(36, 162, 163, 165, 8364, 164)

>>> import array

>>> array.array('I', (ord(symbol) for symbol in symbols))

array('I', [36, 162, 163, 165, 8364, 164])

**Cartesian product in a generator expression: X**

>>> colors = ['black', 'white']

>>> sizes = ['S', 'M', 'L']

>>> for tshirt in ('%s %s' % (c, s) for c in colors for s in sizes):

... print(tshirt)

...

black S

black M

black L

white S

white M

**IMPORTANT:**

Here we are only multiplying 3x3 elements, so it looks like we are not saving much memory, but what if we have two lists with 1000x1000? In this case we use only the memory space necessary for 2000 elements, instead that using 1,000,000(one million) of elements that would be 2M because we are doing cartesian product so it is the double. The memory saved is 2M – 2K = 1,998,000. Practically 2M.

Tuples Are Not Just Immutable Lists:

Some introductory texts about Python present tuples as “immutable lists,” but that is short selling them. Tuples do double duty: they can be used as immutable lists and also as records with no field names. This use is sometimes overlooked, so we will start with that.

Tuples as Records

Tuples hold records: each item in the tuple holds the data for one field and the position of the item gives its meaning. If you think of a tuple just as an immutable list, the quantity and the order of the items may or may not be important, depending on the context. But when using a tuple as a collection of fields, the number of items is often fixed, and their order is always vital. Example 3 shows tuples being used as records. Note that in every expression, sorting the tuple would destroy the information because the meaning of each data item is given by its position in the tuple.

**Example 3 X**

>>> lax\_coordinates = (33.9425, -118.408056)

>>> city, year, pop, chg, area = ('Tokyo', 2003, 32450, 0.66, 8014)

>>> traveler\_ids = [('USA', '31195855'), ('BRA', 'CE342567'),

... ('ESP', 'XDA205856')]

>>> for passport in sorted(traveler\_ids):

... print('%s/%s' % passport)

...

BRA/CE342567

ESP/XDA205856

USA/31195855

>>> for country, \_ in traveler\_ids:

... print(country)

...

USA

BRA

ESP

Tuples Unpacking:

In Example 3, we assigned ('Tokyo', 2003, 32450, 0.66, 8014) to city, year, pop, chg, area in a single statement. Then, in the last line, the % operator assigned each item in the passport tuple to one slot in the format string in the print argument. Those are two examples of tuple unpacking.

The most visible form of tuple unpacking is parallel assignment; that is, assigning items from an iterable to a tuple of variables, as you can see in this example:

>>> lax\_coordinates = (33.9425, -118.408056)

>>> latitude, longitude = lax\_coordinates # tuple unpacking

>>> latitude

33.9425

>>> longitude

-118.408056

>>> b, a = a, b

>>> divmod(20, 8)

(2, 4)

>>> t = (20, 8)

>>> divmod(\*t)

(2, 4)

>>> quotient, remainder = divmod(\*t)

>>> quotient, remainder

(2, 4)

>>> import os

>>> \_, filename = os.path.split('/home/luciano/.ssh/idrsa.pub')

>>> filename

'idrsa.pub'

Using \* to grab excess items

Defining function parameters with \*args to grab arbitrary excess arguments is a classic Python feature. In Python 3, this idea was extended to apply to parallel assignment as well:

>>> a, b, \*rest = range(5)

>>> a, b, rest

(0, 1, [2, 3, 4])

>>> a, b, \*rest = range(3)

>>> a, b, rest

(0, 1, [2])

>>> a, b, \*rest = range(2)

>>> a, b, rest

(0, 1, [])

In the context of parallel assignment, the \* prefix can be applied to exactly one variable, but it can appear in any position:

>>> a, \*body, c, d = range(5)

>>> a, body, c, d

(0, [1, 2], 3, 4)

>>> \*head, b, c, d = range(5)

>>> head, b, c, d

([0, 1], 2, 3, 4)

Nested Tuples Unpacking:

Sometimes we have nested tuples, meaning a tuple inside of another tuple or list, and the process of unpacking means that we create a structure to the right that has the same structure than the structure to the left. We can use this for creating dictionaries, to access a determined value in when using for loop and, for tuple unpacking:

IMPORTANT: This only works when using a list with nested tuples, if is only one we’d have to use normal unpacking using \_\_getitem\_\_()’s normal function [1][0]

metro\_areas = [

 ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)),

 ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),

 ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),

 ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),

 ('Sao Paulo', 'BR', 19.649, (-23.547778, -46.635833)),

]

print('{:15} | {:^9} | {:^9}'.format('', 'lat.', 'long.'))

fmt = '{:15} | {:9.4f} | {:9.4f}'

for name, cc, pop, (latitude, longitude) in metro\_areas:

    if longitude <= 0:

        print(fmt.format(name, latitude, longitude))

**Output X**

| lat. | long.

Mexico City | 19.4333 | -99.1333

New York-Newark | 40.8086 | -74.0204

Sao Paulo | -23.5478 | -46.6358

Named Tuples:

The collections.namedtuple function is a factory that produces subclasses of tuple enhanced with field names and a class name—which helps debugging. Instances of a class that you build with namedtuple take exactly the same amount of memory as tuples because the field names are stored in the class. They use less memory than a regular object because they don’t store attributes in a per-instance \_\_dict\_\_.

**Named Tuples X**

>>> from collections import namedtuple

>>> City = namedtuple('City', 'name country population coordinates')

>>> tokyo = City('Tokyo', 'JP', 36.933, (35.689722, 139.691667))

>>> tokyo

City(name='Tokyo', country='JP', population=36.933, coordinates=(35.689722,

139.691667))

>>> tokyo.population

36.933

>>> tokyo.coordinates

(35.689722, 139.691667)

>>> tokyo[1]

'JP'

Using named tuples is almost as using a class. As we can see you can access their fields by using the dot notation as we do when using a class. Also, we can get their fields by using City.\_fields which is a tuple that is created by default when the namedtuple is create for the first time.

Two parameters are required to create a named tuple: a **class** **name** and a **list** of **field** **names**, which can be given as an iterable of strings or as a single space-delimited string.

Data must be passed as positional arguments to the constructor (in contrast, the tuple constructor takes a single iterable).

You can access the fields by name or position.

More on Named tuples:

A named tuple has some especial attributes, like a normal class. That’s why we think of them like subclasses. Once we have defined the fields of a named tuple, or its attributes in other words, we can just give it new data and build a new one.

Also, we can use various functionalities like see the namedtuple’s fields by using .\_fields, make a new one with .\_make( <iterable> ) and of course, create a new dictionary with the .\_asdict() ,method

Here we can see these three methods in action

**Named tuples especial methods: X**

. . .

>>> City.\_fields

('name', 'country', 'population', 'coordinates')

>>> LatLong = namedtuple('LatLong', 'lat long')

>>> delhi\_data = ('Delhi NCR', 'IN', 21.935, LatLong(28.613889, 77.208889))

>>> delhi = City.\_make(delhi\_data)

>>> delhi.\_asdict()

OrderedDict([('name', 'Delhi NCR'), ('country', 'IN'), ('population',

21.935), ('coordinates', LatLong(lat=28.613889, long=77.208889))])

>>> for key, value in delhi.\_asdict().items():

print(key + ':', value)

name: Delhi NCR

country: IN

population: 21.935

coordinates: LatLong(lat=28.613889, long=77.208889)

Tuples as immutable lists:

We can also use a tuple as an immutable list of items. This is because the tuples accept all the methods in a list that doesn’t include modifying it, like adding or removing items from it. The only exception is the .\_\_reversed\_\_() method which for that case we can use reversed(my\_tuple) and it will work just fine.

Methods and attributes found in list or tuple

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **list** | | **tuple** | |  |
| s.\_\_add\_\_(s2) | | **•** | **•** | s + s2—concatenation | |
| s.\_\_iadd\_\_(s2) | | **•** |  | s += s2—in-place concatenation | |
| s.append(e) | | **•** |  | Append one element after last | |
| s.clear() | | **•** |  | Delete all items | |
| s.\_\_contains\_\_(e) | | **•** | **•** | e in s | |
| s.copy() | | **•** |  | Shallow copy of the list | |
| s.count(e) | | **•** | **•** | Count occurrences of an element | |
| s.\_\_delitem\_\_(p) | | **•** |  | Remove item at position p | |
| s.extend(it) | | **•** |  | Append items from iterable it | |
| s.\_\_getitem\_\_(p) | | **•** | **•** | s[p]—get item at position | |
| s.\_\_getnewargs\_\_() | |  | **•** | Support for optimized serialization with pickle | |
| s.index(e) | | **•** | **•** | Find position of first occurrence of e | |
| s.insert(p, e) | | **•** |  | Insert element e before the item at position p | |
| s.\_\_iter\_\_() | | **•** | **•** | Get iterator | |
| s.\_\_len\_\_() | | **•** | **•** | len(s)—number of items | |
| s.\_\_mul\_\_(n) | | **•** | **•** | s \* n—repeated concatenation | |
| s.\_\_imul\_\_(n) | | **•** |  | s \*= n—in-place repeated concatenation | |
| s.\_\_rmul\_\_(n) | | **•** | **•** | n \* s—reversed repeated concatenation | |
| s.pop([p]) | | **•** |  | Remove and return last item or item at position p | |
| s.remove(e) | | **•** |  | Remove first occurrence of element e by value | |
| s.reverse() | | **•** |  | Reverse the order of the items in place | |
| s.\_\_reversed\_\_() | | **•** |  | Get iterator to scan items from last to first | |
| s.\_\_setitem\_\_(p, e) | | **•** |  | s[p] = e—put e in position p, overwriting item | |
| s.sort([key], [reverse]) | | **•** |  | Sort items in place with optional keyword arguments key and reverse | |

Slicing:

Slicing is a very common feature used by many programmers using python and is this day-to-day use what make us forget how powerful this feature is. Slicing in python excludes the last item, this is in slicing and also in the range function. This is because is of course we are dealing with computers and computers start counting from zero. Some convenient features of this way of doing slicing are:

• It’s easy to see the length of a slice or range when only the stop position is given:

range(3) and my\_list[:3] both produce three items.

• It’s easy to compute the length of a slice or range when start and stop are given, just

subtract stop - start.

• It’s easy to split a sequence in two parts at any index x, without overlapping: simply

get my\_list[:x] and my\_list[x:]. For example:

**Slicing: X**

>>> l = [10, 20, 30, 40, 50, 60]

>>> l[:2] # split at 2

[10, 20]

>>> l[2:]

[30, 40, 50, 60]

>>> l[:3] # split at 3

[10, 20, 30]

>>> l[3:]

[40, 50, 60]

Slice objects:

This is no secret, but worth repeating just in case: s[a:b:c] can be used to specify a stride or step c, causing the resulting slice to skip items. The stride can also be negative, returning items in reverse.

Three examples make this clear:

**Named tuples especial methods: X**

. . .

>>> s = 'bicycle'

>>> s[::3]

'bye'

>>> s[::-1]

'elcycib'

>>> s[::-2]

'eccb'

The notation a:b:c is only valid within [] when used as the indexing or subscript operator, and it produces a slice object: slice(a, b, c).To evaluate the expression seq[start:stop:step], Python calls seq.\_\_getitem\_\_(slice(start, stop, step)). Even if you are not implementing your own sequence types, knowing about slice objects is useful because it lets you assign names to slices, just like spreadsheets allow naming of cell ranges.

Example:

Suppose you need to parse flat-file data like the invoice shown in Example 2-11. Instead of filling your code with hardcoded slices, you can name them. See how readable this makes the for loop at the end of the example.

invoice = """

0.....6.................................40........52...55........

1909 Pimoroni PiBrella $17.50 3 $52.50

1489 6mm Tactile Switch x20 $4.95 2 $9.90

1510 Panavise Jr. - PV-201 $28.00 1 $28.00

1601 PiTFT Mini Kit 320x240 $34.95 1 $34.95

"""

SKU = slice(0, 6)

DESCRIPTION = slice(6, 40)

UNIT\_PRICE = slice(40, 52)

QUANTITY = slice(52, 55)

ITEM\_TOTAL = slice(55, None)

line\_items = invoice.split('\n')[2:]

for item in line\_items:

    print(item[UNIT\_PRICE], item[DESCRIPTION])

**Output: X**

$17.50 Pimoroni PiBrella

$4.95 6mm Tactile Switch x20

$28.00 Panavise Jr. - PV-201

$34.95 PiTFT Mini Kit 320x240

Building Lists of Lists:

There are sometimes when we need to make nested lists, meaning a list inside of a list to represent something. We must be very careful when doing this kind of operations because it can backfire. The deal here is that when we create a list, this list is not containing the actual element, but referencing it. This means that if we make a list of the same element repeated lots of times, we’ll have a list full of the same object. So, what’s the inconvenient? The bad thing here is that the list is always pointing to the same object in memory, this means that if we modify that object, then the entire list will change, because all the elements are referencing that unique object:

**Output: X**

>>> weird\_board = [['\_']\*3]\*3

>>> weird\_board

[['\_', '\_', '\_'], ['\_', '\_', '\_'], ['\_', '\_', '\_']]

>>> weird\_board[1][2] = '0'

>>> weird\_board

[['\_', '\_', '0'], ['\_', '\_', '0'], ['\_', '\_', '0']]

>>>

We can fix this by using a for loop instead of just multiplying:

**Output: X**

>>> board = []

>>> for i in range(3):

... row = ['\_'] \* 3 #

... board.append(row)

...

>>> board

[['\_', '\_', '\_'], ['\_', '\_', '\_'], ['\_', '\_', '\_']]

>>> board[2][0] = 'X'

>>> board #

[['\_', '\_', '\_'], ['\_', '\_', '\_'], ['X', '\_', '\_']]

>>>

Augmented Assignment with Sequences:

The augmented assignment operators += and \*= behave very differently depending on the first operand. To simplify the discussion, we will focus on augmented addition first (+=), but the concepts also apply to \*= and to other augmented assignment operators. The special method that makes += work is \_\_iadd\_\_ (for “in-place addition”). However, if \_\_iadd\_\_ is not implemented, Python falls back to calling \_\_add\_\_. Consider this simple expression: a += b.

If a implements \_\_iadd\_\_, that will be called. In the case of mutable sequences (e.g., list, bytearray, array.array), a will be changed in place (i.e., the effect will be like a.extend(b)). However, when a does not implement \_\_iadd\_\_, the expression a += b has the same effect as a = a + b: the expression a + b is evaluated first, producing a new object, which is then bound to a. In other words, the identity of the object bound to a may or may not change, depending on the availability of \_\_iadd\_\_. In general, for mutable sequences, it is a good bet that \_\_iadd\_\_ is implemented and that += happens in place. For immutable sequences, clearly there is no way for that to happen.

**Output: X**

>>> l = [1, 2, 3]

>>> id(l)

4311953800

>>> l \*= 2

>>> l

[1, 2, 3, 1, 2, 3]

>>> id(l)

4311953800

>>> t = (1, 2, 3)

>>> id(t)

4312681568

>>> t \*= 2

>>> id(t)

4301348296

>>>

**Exception:**

str is an exception to this description. Because string building with += in loops is so common in the wild, CPython is optimized for this use case. str instances are allocated in memory with room to spare, so that concatenation does not require copying the whole string every time.

A += Assignment Puzzler:

Let’s picture something. Let’s say we are getting ourselves a tuple, but inside of that tuple we have a mutable object, like list for example, which is a reference to the object that is carrying. Here is the question:

What would happen if we tried to assign a new value to the list inside the tuple?

>>> t = (1, 2, [30, 40])

>>> t[2] += [50, 60]

What happens next? Choose the best answer:

a. t becomes (1, 2, [30, 40, 50, 60]).

b. TypeError is raised with the message 'tuple' object does not support item assignment.

c. Neither.

d. Both a and b.

When I saw this, I was pretty sure the answer was b, but it’s actually , **“Both a and b.”**!

The unexpected result: item t2 is changed and an exception is raised

>>> t = (1, 2, [30, 40])

>>> t[2] += [50, 60]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

>>> t

(1, 2, [30, 40, 50, 60])

**Explanation:**

What is happening here is that Python can actually change the content of the list just because the list is a mutable object. It has nothing to do with if the list is part of some other structure or not. The problem comes when it is part of a data structure like a tuple where it’s items must not be changed.

Here the python interpreter first goes to the referenced item and try the operation, since the operation is successful the content of the list is changed. But then it tell the tuple that the content has been modified, and here is where we get our error.

If the order that python does this was the other way around, the error would get in the middle and we wouldn’t be able to change it by referencing the object in the tuple, but we could change it anyways by referencing it in the list, or by changing the actual object, if the object is mutable which would be the same effect.

Bytecode for the expression s[a] +=b :

**Bytecode: X**

>>> dis.dis('s[a] += b')

0 LOAD\_NAME 0 (s)

 3 LOAD\_NAME 1 (a)

 6 DUP\_TOP\_TWO

 7 BINARY\_SUBSCR

 8 LOAD\_NAME 2 (b)

 11 INPLACE\_ADD

 12 ROT\_THREE

 13 STORE\_SUBSCR

 14 LOAD\_CONST 0 (None)

 17 RETURN\_VALUE

**I take three lessons from this:**

• Putting mutable items in tuples is not a good idea.

• Augmented assignment is not an atomic operation—we just saw it throwing an exception after doing part of its job.

• Inspecting Python bytecode is not too difficult and is often helpful to see what is going on under the hood. After witnessing the subtleties of using + and \* for concatenation, we can change the subject to another essential operation with sequences: sorting

list.sort and sorted Built-in function:

The list.sort method sorts a list in place—that is, without making a copy. It returns None to remind us that it changes the target object and does not create a new list. This is an important Python API convention: functions or methods that change an object in place should return None to make it clear to the caller that the object itself was changed, and no new object was created. The same behavior can be seen, for example, in the random.shuffle function.

The convention of returning None to signal in-place changes has a drawback: you cannot cascade calls to those methods. In contrast, methods that return new objects (e.g., all str methods) can be cascaded in the fluent interface style.

Positional arguments for **list.sort** and **list.sorted** :

reverse

If True, the items are returned in descending order (i.e., by reversing the comparison of the items). The default is False.

key

A one-argument function that will be applied to each item to produce its sorting key. For example, when sorting a list of strings, key=str.lower can be used to perform a case-insensitive sort, and key=len will sort the strings by character length. The default is the identity function (i.e., the items themselves are compared). The key optional keyword parameter can also be used with the min() and max() built-ins and with other functions from the standard library (e.g., itertools.groupby() and heapq.nlargest()).

**Sorting: X**

>>> fruits = ['grape', 'raspberry', 'apple', 'banana']

>>> sorted(fruits)

['apple', 'banana', 'grape', 'raspberry']

>>> fruits

['grape', 'raspberry', 'apple', 'banana']

>>> sorted(fruits, reverse=True)

['raspberry', 'grape', 'banana', 'apple']

>>> sorted(fruits, key=len)

['grape', 'apple', 'banana', 'raspberry']

>>> sorted(fruits, key=len, reverse=True)

['raspberry', 'banana', 'grape', 'apple']

>>> fruits

['grape', 'raspberry', 'apple', 'banana']

>>> fruits.sort()

>>> fruits

['apple', 'banana', 'grape', 'raspberry']

Managing Ordered Sequences with bisect:

The bisect module offers two main functions—bisect and insort—that use the binary search algorithm to quickly find and insert items in any sorted sequence.

Searching with bisect:

bisect(haystack, needle) does a binary search for needle in haystack—which must be a sorted sequence—to locate the position where needle can be inserted while maintaining haystack in ascending order. In other words, all items appearing up to that position are less than or equal to needle. You could use the result of bisect(haystack, needle) as the index argument to haystack.insert(index, needle) however, using insort does both steps, and is faster.

Also bisect can take two optional arguments, lo and hi. As their names indicate, this are the values between the bisect function will be used. Basically, this means that bisect will be used in the slice of the sequence that is defined between lo and hi. By default, lo is set to lo = 0 and hi is set to

hi = len(<sequence>)

Bisect left and bisect right :

When we use bisect, we are actually referring to bisect\_right. This function has a sister function named bisect\_left. The difference between these two functions is not that they sort or search from lest to right or from right to left, but where does the bisect is done, to the very index that has been found, or after the index. These behavior can only be spotted when the needle is equal to the haystack. Therefore, the bisect\_right function insert the needle for the search at the very position of the item, and the bisect\_left function insert the needle right before, to the left.

Visualization of bisect\_left and bisect\_right :

For the visualization of how the work do we’ll use the following script for bisect and show the output for the bisect\_left and bisect\_right in the next page so we can have a visual of this behavior.

Code for bisect\_right and its representation :

import bisect

import sys

HAYSTACK = [1, 4, 5, 6, 8, 12, 15, 20, 21, 23, 23, 26, 29, 30]

NEEDLES = [0, 1, 2, 5, 8, 10, 22, 23, 29, 30, 31]

ROW\_FMT = '{0:2d} @ {1:2d} {2}{0:<2d}'

def demo(bisect\_fn):

    for needle in reversed(NEEDLES):

        position = bisect\_fn(HAYSTACK, needle)

        offset = position \* ' |'

        print(ROW\_FMT.format(needle, position, offset))

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

    if sys.argv[-1] == 'left':

        bisect\_fn = bisect.bisect\_left

    else:

        bisect\_fn = bisect.bisect

print('DEMO:', bisect\_fn.\_\_name\_\_)

print('haystack ->', ' '.join('%2d' % n for n in HAYSTACK))

demo(bisect\_fn)

Code for bisect\_right and its representation :

**Output : X**

>>>

DEMO: bisect\_right

haystack -> 1 4 5 6 8 12 15 20 21 23 23 26 29 30

31 @ 14 | | | | | | | | | | | | | |31

30 @ 14 | | | | | | | | | | | | | |30

29 @ 13 | | | | | | | | | | | | |29

23 @ 11 | | | | | | | | | | |23

22 @ 9 | | | | | | | | |22

10 @ 5 | | | | |10

8 @ 5 | | | | |8

5 @ 3 | | |5

2 @ 1 |2

1 @ 1 |1

0 @ 0 0

>>>

Code for bisect\_left and its representation :

**Output : X**

>>>

DEMO: bisect\_left

haystack -> 1 4 5 6 8 12 15 20 21 23 23 26 29 30

31 @ 14 | | | | | | | | | | | | | |31

30 @ 14 | | | | | | | | | | | | |30

29 @ 13 | | | | | | | | | | | |29

23 @ 11 | | | | | | | | |23

22 @ 9 | | | | | | | | |22

10 @ 5 | | | | |10

8 @ 5 | | | |8

5 @ 3 | |5

2 @ 1 |2

1 @ 1 1

0 @ 0 0

>>>

Inserting with bisect.insort:

Sorting our sequences can be very expensive, resourcefully talking. Sort our sequence every time requires us, in the wort case scenario, to go through all the items in our sequence, which might easily be 1M elements. Therefore, when we sort out sequence, we want to keep it sorted when introducing new items so we can still use the binary search to get an item from the sequence.

We can easily deduce, that once our sequence is sorted is very easy to insert an element into it by using the same method we use to search an item in it, and then keep it sorted. But the bisect module does this for us with the insort function.

Code for bisect.insort :

import bisect

import random

SIZE = 7

random.seed(1729)

my\_list = []

for i in range(SIZE):

    new\_item = random.randrange(SIZE\*2)

    bisect.insort(my\_list, new\_item)

    print('%2d ->' % new\_item, my\_list)

Representation and output for bisect.insort :

**Output : X**

>>>

11 -> [11, 12]

11 -> [11, 11, 12]

13 -> [11, 11, 12, 13]

5 -> [5, 11, 11, 12, 13]

10 -> [5, 10, 11, 11, 12, 13]

3 -> [3, 5, 10, 11, 11, 12, 13]

10 -> [3, 5, 10, 10, 11, 11, 12, 13]

>>>

When a list is not the answer :

As we have seen in this document, lists are not always the best option for every case and every circumstance. They are very different from other types of sequences, array sequences. A tuple for example takes less space in memory, this is because they take the data in the actual field. There are other types of array-type sequences that are better to use when we are in determined situations, like saving space, using a lot of data, LIFO, and FIFO structures, meaning queues. Therefore, now we’ll cover the structures that make this possible.

Also, if we are doing lots of checks in our structure, meaning checking if the item exists in the sequence, is better to use another data structure, a set. But sets are not sequences because their items are not ordered. We’ll see this in a document that is only dedicated to sets and dictionaries.

Other data structures and main advantages :

array.array

memoryview

NumPy and SciPy

Deques and other Queues

Arrays :

A Python array as is learned, is actually a C array. The sequenced data structure called array does not come with python by default, but it comes in the standard library, and we can use it very easy. We refer to it as array.array because this is an object called array, that comes from the array module. The array.array sequence supports most of the methods that any mutable sequence does, plus some other super helpful methods that are unique to this data structure.

Since an array.array is a C array, we must let Python know how much space is going to take each item in the array, meaning how big is the number of the object going=g to be. This looks annoying but is actually one of the best functionalities because saves a lot of space. If you create an array('b'), then each item will be stored in a single byte and interpreted as an integer from –128 to 127. For large sequences of numbers, this saves a lot of memory. And Python will not let you put any number that does not match the type for the array.

Creating, saving, and loading a large array of floats:

**Output : X**

>>>

>>> from array import array

>>> from random import random

>>> floats = array('d', (random() for i in range(10\*\*7)))

>>> floats[-1]

0.07802343889111107

>>> fp = open('floats.bin', 'wb')

>>> floats.tofile(fp)

>>> fp.close()

>>> floats2 = array('d')

>>> fp = open('floats.bin', 'rb')

>>> floats2.fromfile(fp, 10\*\*7)

>>> fp.close()

>>> floats2[-1]

0.07802343889111107

>>> floats2 == floats

True

>>>

Benefits of using a C array :

As you can see, array.tofile and array.fromfile are easy to use. If you try the ex‐ ample, you’ll notice they are also very fast. A quick experiment show that it takes about 0.1s for array.fromfile to load 10 million double-precision floats from a binary file created with array.tofile. That is nearly 60 times faster than reading the numbers from a text file, which also involves parsing each line with the float built-in. Saving with array.tofile is about 7 times faster than writing one float per line in a text file. In addition, the size of the binary file with 10 million doubles is 80,000,000 bytes (8 bytes per double, zero overhead), while the text file has 181,515,739 bytes, for the same data.

Methods and attributes found in list or array

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **list** | | **array** | |  |
| s.\_\_add\_\_(s2) | | **•** | **•** | s + s2—concatenation | |
| s.\_\_iadd\_\_(s2) | | **•** | **•** | s += s2—in-place concatenation | |
| s.append(e) | | **•** | **•** | Append one element after last | |
| s.byteswap() | |  | **•** | Swap bytes of all items in array for endianness conversion | |
| s.clear() | | **•** |  | Delete all items | |
| s.\_\_contains\_\_(e) | | **•** | **•** | e in s | |
| s.copy() | | **•** |  | Shallow copy of the list | |
| s.\_\_copy\_\_() | |  | **•** | Support for copy.copy | |
| s.count(e) | | **•** | **•** | Count occurrences of an element | |
| s.\_\_delitem\_\_(p) | | **•** | **•** | Remove item at position p | |
| s.extend(it) | | **•** | **•** | Append items from iterable it | |
| s.frombytes(b) | |  | **•** | Append items from byte sequence as packed machine values | |
| s.fromfile(f, n) | |  | **•** | Append n items from binary file f interpreted as packed machine values | |
| s.fromlist(l) | |  | **•** | Append items from list; if one causes TypeError, none are appended | |
| s.\_\_getitem\_\_(p) | | **•** | **•** | s[p]—get item at position | |
| s.index(e) | | **•** | **•** | Find position of first occurrence of e | |
| s.insert(p, e) | | **•** | **•** | Insert element e before the item at position p | |
| s.itemsize | |  | **•** | Length in bytes of each array item | |
| s.\_\_iter\_\_() | | **•** | **•** | Get iterator | |
| s.\_\_len\_\_() | | **•** | **•** | len(s)—number of items | |
| s.\_\_mul\_\_(n) | | **•** | **•** | s \* n—repeated concatenation | |
| s.\_\_imul\_\_(n) | | **•** | **•** | s \*= n—in-place repeated concatenation | |
| s.\_\_rmul\_\_(n) | | **•** | **•** | n \* s—reversed repeated concatenation | |
| s.pop([p]) | | **•** | **•** | Remove and return last item or item at position p | |
| s.remove(e) | | **•** | **•** | Remove first occurrence of element e by value | |
| s.reverse() | | **•** | **•** | Reverse the order of the items in place | |
|  | |  |  |  | |

Methods and attributes found in list or array (Cont):

=

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | |  |  |  |
| s.\_\_reversed\_\_() | | **•** |  | Get iterator to scan items from last to first |
| s.\_\_setitem\_\_(p, e) | | **•** | **•** | s[p] = e—put e in position p, overwriting item |
| s.sort([key], [reverse]) | | **•** |  | Sort items in place with optional keyword arguments key and reverse |
| s.tobytes() |  | | **•** | Return items as packed machine values in a byte object |
| s.tofile(f) |  | | **•** | Save items as packed machine values to binary file f |
| s.tolist() |  | | **•** | Return items as numeric objects in a list |
| s.typecode |  | | **•** | One-character string identifying the C type of the items |

Memoryviews :

This built-in class named memoryview, is a shared sequence type that allows you to handle slices of array without actually having to copy them. This means that this structure let you see and modify an object that shares memory with another one. This behavior comes from the external module NumPy.

*A memoryview is essentially a generalized NumPy array structure in Python itself (without the math). It allows you to share memory between data-structures (things like PIL images, SQLlite databases, NumPy arrays, etc.) without first copying. This is very important for large data sets.*

***Travis Oliphant, lead author of Num‐ Py***

Using notation similar to the array module, the memoryview.cast method lets you change the way multiple bytes are read or written as units without moving bits around (just like the C cast operator). memoryview.cast returns yet another memoryview object, always sharing the same memory.

Changing the value of an array item by poking one of its bytes:

**Output : X**

>>>

>>> numbers = array.array('h', [-2, -1, 0, 1, 2])

>>> memv = memoryview(numbers)

>>> len(memv)

5

>>> memv[0]

-2

>>> memv\_oct = memv.cast('B')

>>> memv\_oct.tolist()

[254, 255, 255, 255, 0, 0, 1, 0, 2, 0]

>>> memv\_oct[5] = 4

>>> numbers

array('h', [-2, -1, 1024, 1, 2])

>>>

NumPy and SciPy :

NumPy is not a canonic package, is not included in the python standard library but is considered one of the most important python modules because NumPy (mainly) and SciPy are the ones who opened the doors of the scientific applications of python. In the case os NumPy this is because it works with matrices, a very important component in mathematics and physics problem, and supports all of the operations that are to be made with these structures.

SciPy is a library, written on top of NumPy, offering many scientific computing algorithms from linear algebra, numerical calculus, and statistics. SciPy is fast and reliable because it leverages the widely used C and Fortran code base from the Netlib Repository. In other words, SciPy gives scientists the best of both worlds: an interactive prompt and high-level Python APIs, together with industrial-strength number-crunching functions optimized in C and Fortran.

Basic operations with rows and columns in a numpy.ndarray:

**Output : X**

>>> import numpy

>>> a = numpy.arange(12)

>>> a

array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

>>> type(a)

<class 'numpy.ndarray'>

>>> a.shape

(12,)

>>> a.shape = 3, 4

>>> a

array([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11]])

>>> a[2]

array([ 8, 9, 10, 11])

>>> a[2, 1]

9

>>> a[:, 1]

array([1, 5, 9])

>>> a.transpose()

array([[ 0, 4, 8],

[ 1, 5, 9],

[ 2, 6, 10],

[ 3, 7, 11]])

>>>

High-level operations: Loading, Saving, and Operating on numpy.ndarray:

**Output : X**

>>>

>>> import numpy

>>> floats = numpy.loadtxt('floats-10M-lines.txt')

>>> floats[-3:]

array([ 3016362.69195522, 535281.10514262, 4566560.44373946])

>>> floats \*= .5

>>> floats[-3:]

array([ 1508181.34597761, 267640.55257131, 2283280.22186973])

>>> from time import perf\_counter as pc

>>> t0 = pc(); floats /= 3; pc() - t0

0.03690556302899495

>>> numpy.save('floats-10M', floats)

>>> floats2 = numpy.load('floats-10M.npy', 'r+')

>>>floats2[-3:]

memmap([ 3016362.69195522, 535281.10514262, 4566560.44373946])ts2 \*= 6

>>>

About NumPy and SciPy :

NumPy and SciPy are vast libraries, and we can’t really cover them in a couple of pages, so this is just an appetizer. These libraries are the foundation of other awesome tools such as the Pandas and Blaze data analysis libraries, which provide efficient array types that can hold nonnumeric data as well as import/export functions compatible with many different formats (e.g., .csv, .xls, SQL dumps, HDF5, etc.).

Deques and other queues:

The .append and .pop methods make a list usable as a stack or a queue (if you use .append and .pop(0), you get LIFO behavior). But inserting and removing from the left of a list (the 0-index end) is costly because the entire list must be shifted. The class collections.deque is a thread-safe double-ended queue designed for fast inserting and removing from both ends. It is also the way to go if you need to keep a list of “last seen items” or something like that, because a deque can be bounded—i.e., created with a maximum length—and then, when it is full, it discards items from the opposite end when you append new ones

Working with a deque:

**Output : X**

>>>

>>> from collections import deque

>>> dq = deque(range(10), maxlen=10)

>>> dq

deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)

>>> dq.rotate(3)

>>> dq

deque([7, 8, 9, 0, 1, 2, 3, 4, 5, 6], maxlen=10)

>>> dq.rotate(-4)

>>> dq

deque([1, 2, 3, 4, 5, 6, 7, 8, 9, 0], maxlen=10)

>>> dq.appendleft(-1)

>>> dq

deque([-1, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)

>>> dq.extend([11, 22, 33])

>>> dq

deque([3, 4, 5, 6, 7, 8, 9, 11, 22, 33], maxlen=10)

>>> dq.extendleft([10, 20, 30, 40])

>>> dq

deque([40, 30, 20, 10, 3, 4, 5, 6, 7, 8], maxlen=10)

>>>

Methods and attributes found in list or deque:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **list** | | **deque** | |  |
| s.\_\_add\_\_(s2) | | **•** |  | s + s2—concatenation | |
| s.\_\_iadd\_\_(s2) | | **•** | **•** | s += s2—in-place concatenation | |
| s.append(e) | | **•** | **•** | Append one element to the right (after last) | |
| s.appendleft(e) | |  | **•** | Append one element to the left (before first) | |
| s.clear() | | **•** | **•** | Delete all items | |
| s.\_\_contains\_\_(e) | | **•** |  | e in s | |
| s.copy() | | **•** |  | Shallow copy of the list | |
| s.\_\_copy\_\_() | |  | **•** | Support for copy.copy (shallow copy) | |
| s.count(e) | | **•** | **•** | Count occurrences of an element | |
| s.\_\_delitem\_\_(p) | | **•** | **•** | Remove item at position p | |
| s.extend(it) | | **•** | **•** | Append items from iterable it to the right | |
| s.extendleft(i) | | **•** |  | Append items from iterable it to the left | |
| s.\_\_getitem\_\_(p) | | **•** | **•** | s[p]—get item at position | |
| s.index(e) | | **•** |  | Find position of first occurrence of e | |
| s.insert(p, e) | | **•** |  | Insert element e before the item at position p | |
| s.\_\_iter\_\_() | | **•** | **•** | Get iterator | |
| s.\_\_len\_\_() | | **•** | **•** | len(s)—number of items | |
| s.\_\_mul\_\_(n) | | **•** |  | s \* n—repeated concatenation | |
| s.\_\_imul\_\_(n) | | **•** |  | s \*= n—in-place repeated concatenation | |
| s.\_\_rmul\_\_(n) | | **•** |  | n \* s—reversed repeated concatenation | |
| s.pop() | | **•** | **•** | Remove and return last item | |
| s.popleft() | |  | **•** | Remove and return first item | |
| s.remove(e) | | **•** | **•** | Remove first occurrence of element e by value | |
| s.reverse() | | **•** | **•** | Reverse the order of the items in place | |
| s.\_\_reversed\_\_() | | **•** | **•** | Get iterator to scan items from last to first | |
| s.rotate(n) | |  | **•** | Move n items from one end to the other | |
| s.\_\_setitem\_\_(p, e) | | **•** |  | s[p] = e—put e in position p, overwriting item | |
| s.sort([key], [reverse]) | | **•** |  | Sort items in place with optional keyword arguments key and reverse | |

Summary:

The Python sequences:

Python sequences can be categorized in mutable, and immutable, but is also useful to consider other type of criterion. This is if they actually contain the element, or if they reference it. We call those who contain the element: flat sequences, and those who reference the position of the element in memory, container sequences

List comprehensions and generator objects:

These are powerful notations to build and initialize sequences. Listcomps actually create a list in place while the genexp they yield an element per iteration, saving tons of memory and resources.

Tuples and Named Tuples:

Tuples have an apparent limitation and is that they won’t allow you to change the values once they’re set. This is actually a feature, when we want to deal with an immutable list, or to save records that won’t change in time. Named tuples are not so new, but deserve more attention: like tuples, they have very little overhead per instance, yet provide convenient access to the fields by name and a handy .\_asdict() to export the record as an OrderedDict.

Sequence Slicing:

This is a classic python syntax feature, and it can be really powerful once we understand what is going on under the hood. Multidimensional slicing and ellipsis (...) notation, as used in NumPy, may also be supported by user-defined sequences. Assigning to slices is a very expressive way of editing mutable sequences.

Augmented Assignment and repeated concatenation:

Repeated concatenation can be a very useful feature like for example in <sequence> \* n but we have to watch out for same object referencing in a structure. Augmented assignment with += and \*= behaves differently for mutable and immutable sequences. In the latter case, these operators necessarily build new sequences. But if the target sequence is mutable, it is usually changed in place—but not always, depending on how the sequence is implemented.

Sorting and bisecting:

The sort method and the sorted built-in function are easy to use and flexible, thanks to the key optional argument they accept, with a function to calculate the ordering criterion. By the way, key can also be used with the min and max built-in functions. To keep a sorted sequence in order, always insert items into it using bisect.insort; to search it efficiently, use bisect.bisect.