List

Fork me on GitHub

The list is a common data structure which we use to store objects. Most of the time, programmers concern about getting, setting, searching, filtering, and sorting. Furthermore, sometimes, we waltz ourself into common pitfalls of the memory management. Thus, the main goal of this cheat sheet is to collect some common operations and pitfalls.

From Scratch

There are so many ways that we can manipulate lists in Python. Before we start to learn those versatile manipulations, the following snippet shows the most common operations of lists.

```
>>> a = [1, 2, 3, 4, 5]
>>> # contains
>>> 2 in a
True
>>> # positive index
>>> a[0]
1
>>> # negative index
>>> a[-1]
5
>>> # slicing list[start:end:step]
>>> a[1:]
[2, 3, 4, 5]
>>> a[1:-1]
[2, 3, 4]
>>> a[1:-1:2]
[2, 4]
>>> # reverse
>>> a[::-1]
[5, 4, 3, 2, 1]
>>> a[:0:-1]
[5, 4, 3, 2]
>>> # set an item
>>> a[0] = 0
>>> a
[0, 2, 3, 4, 5]
>>> # append items to list
>>> a.append(6)
>>> a
[0, 2, 3, 4, 5, 6]
>>> a.extend([7, 8, 9])
>>> a
[0, 2, 3, 4, 5, 6, 7, 8, 9]
>>> # delete an item
>>> del a[-1]
>>> a
[0, 2, 3, 4, 5, 6, 7, 8]
>>> # list comprehension
>>> b = [x for x in range(3)]
>>> b
[0, 1, 2]
>>> # add two lists
>>> a + b
[0, 2, 3, 4, 5, 6, 7, 8, 0, 1, 2]
```

Initialize

Generally speaking, we can create a list through * operator if the item in the list expression is an immutable object.

```
>>> a = [None] * 3
>>> a
[None, None, None]
>>> a[0] = "foo"
>>> a
['foo', None, None]
```

However, if the item in the list expression is a mutable object, the `*` operator will copy the reference of the item N times. In order to avoid this pitfall, we should use a list comprehension to initialize a list.

```
>>> a = [[]] * 3
>>> b = [[] for _ in range(3)]
>>> a[0].append("Hello")
>>> a
[['Hello'], ['Hello'], ['Hello']]
>>> b[0].append("Python")
>>> b
[['Python'], [], []]
```

Copy

Assigning a list to a variable is a common pitfall. This assignment does not copy the list to the variable. The variable only refers to the list and increase the reference count of the list.

```
import sys
>>> a = [1, 2, 3]
>>> sys.getrefcount(a)
2
>>> b = a
>>> sys.getrefcount(a)
3
>>> b[2] = 123456  # a[2] = 123456
>>> b
[1, 2, 123456]
>>> a
[1, 2, 123456]
```

There are two types of copy. The first one is called *shallow copy* (non-recursive copy) and the second one is called *deep copy* (recursive copy). Most of the time, it is sufficient for us to copy a list by shallow copy. However, if a list is nested, we have to use a deep copy.

```
>>> # shallow copy
>>> a = [1, 2]
>>> b = list(a)
>>> b[0] = 123
>>> a
[1, 2]
>>> b
[123, 2]
>>> a = [[1], [2]]
>>> b = list(a)
>>> b[0][0] = 123
>>> a
[[123], [2]]
>>> b
[[123], [2]]
>>> # deep copy
>>> import copy
>>> a = [[1], [2]]
>>> b = copy.deepcopy(a)
>>> b[0][0] = 123
>>> a
[[1], [2]]
>>> b
[[123], [2]]
```

Using `slice`

Sometimes, our data may concatenate as a large segment such as packets. In this case, we will represent the range of data by using `slice` objects as explaining variables instead of using *slicing expressions*.

```
>>> icmp = (
...     b"080062988e2100005bff49c20005767c"
...     b"08090a0b0c0d0e0f1011121314151617"
...     b"18191a1b1c1d1e1f2021222324252627"
...     b"28292a2b2c2d2e2f3031323334353637"
... )
>>> head = slice(0, 32)
>>> data = slice(32, len(icmp))
>>> icmp[head]
b'080062988e2100005bff49c20005767c'
```

List Comprehensions

List comprehensions which was proposed in PEP 202 provides a graceful way to create a new list based on another list, sequence, or some object which is iterable. In addition, we can use this expression to substitute `map` and `filter` sometimes.

```
>>> [x for x in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [(lambda x: x**2)(i) for i in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [x for x in range(10) if x > 5]
[6, 7, 8, 9]
>>> [x if x > 5 else 0 for x in range(10)]
[0, 0, 0, 0, 0, 0, 6, 7, 8, 9]
>>> [x + 1 if x < 5 else x + 2 if x > 5 else x + 5 for x in range(10)]
[1, 2, 3, 4, 5, 10, 8, 9, 10, 11]
>>> [(x, y) for x in range(3) for y in range(2)]
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]
```

Unpacking

Sometimes, we want to unpack our list to variables in order to make our code become more readable. In this case, we assign N elements to N variables as following example.

```
>>> arr = [1, 2, 3]
>>> a, b, c = arr
>>> a, b, c
(1, 2, 3)
```

Based on PEP 3132, we can use a single asterisk to unpack N elements to the number of variables which is less than N in Python 3.

```
>>> arr = [1, 2, 3, 4, 5]
>>> a, b, *c, d = arr
>>> a, b, d
(1, 2, 5)
>>> c
[3, 4]
```

Using `enumerate`

`enumerate` is a built-in function. It helps us to acquire indexes (or a count) and elements at the same time without using `range(len(list))`. Further information can be found on Looping Techniques.

```
>>> for i, v in enumerate(range(3)):
...     print(i, v)
...
0 0
1 1
2 2
>>> for i, v in enumerate(range(3), 1): # start = 1
...     print(i, v)
...
1 0
2 1
3 2
```

Zip Lists

zip enables us to iterate over items contained in multiple lists at a time. Iteration stops whenever one of the lists is exhausted. As a result, the length of the iteration is the same as the shortest list. If this behavior is not desired, we can use `itertools.zip_longest` in **Python 3** or `itertools.izip_longest` in **Python 2**.

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> list(zip(a, b))
[(1, 4), (2, 5), (3, 6)]
>>> c = [1]
>>> list(zip(a, b, c))
[(1, 4, 1)]
>>> from itertools import zip_longest
>>> list(zip_longest(a, b, c))
[(1, 4, 1), (2, 5, None), (3, 6, None)]
```

Filter Items

filter is a built-in function to assist us to remove unnecessary items. In **Python 2**, `filter` returns a list. However, in **Python 3**, `filter` returns an *iterable object*. Note that *list comprehension* or *generator expression* provides a more concise way to remove items.

```
>>> [x for x in range(5) if x > 1]
[2, 3, 4]
>>> l = ['1', '2', 3, 'Hello', 4]
>>> f = lambda x: isinstance(x, int)
>>> filter(f, l)
<filter object at 0x10bee2198>
>>> list(filter(f, l))
[3, 4]
>>> list((i for i in l if f(i)))
[3, 4]
```

Stacks

There is no need for an additional data structure, stack, in Python because the `list` provides `append` and `pop` methods which enable us use a list as a stack.

```
>>> stack = []
>>> stack.append(1)
>>> stack.append(2)
>>> stack.append(3)
>>> stack
[1, 2, 3]
>>> stack.pop()
3
>>> stack.pop()
2
>>> stack
[1]
```

`in` Operation

We can implement the `__contains__` method to make a class do `in` operations. It is a common way for a programmer to emulate a membership test operations for custom classes.

```
class Stack:

    def __init__(self):
        self.__list = []

    def push(self, val):
        self.__list.append(val)
    def pop(self):
        return self.__list.pop()

    def __contains__(self, item):
        return True if item in self.__list else False

stack = Stack()
stack.push(1)
print(1 in stack)
print(0 in stack)
```

Example

```
python stack.py
True
False
```

Accessing Items

Making custom classes perform get and set operations like lists is simple. We can implement a `__getitem__` method and a `__setitem__` method to enable a class to retrieve and overwrite data by index. In addition, if we want to use the function, `len`, to calculate the number of elements, we can implement a `__len__` method.

```
class Stack:

    def __init__(self):
        self.__list = []

    def push(self, val):
        self.__list.append(val)

    def pop(self):
        return self.__list.pop()

    def __repr__(self):
        return "{}".format(self.__list)

    def __len__(self):
        return len(self.__list)

    def __getitem__(self, idx):
        return self.__list[idx]

    def __setitem__(self, idx, val):
        self.__list[idx] = val


stack = Stack()
stack.push(1)
stack.push(2)
print("stack:", stack)

stack[0] = 3
print("stack:", stack)
print("num items:", len(stack))
```

Example

```
$ python stack.py
stack: [1, 2]
stack: [3, 2]
num items: 2
```

Delegating Iterations

If a custom container class holds a list and we want iterations to work on the container, we can implement a `__iter__` method to delegate iterations to the list. Note that the method, `__iter__`, should return an *iterator object*, so we cannot return the list directly; otherwise, Python raises a `TypeError`.

```
class Stack:

    def __init__(self):
        self.__list = []

    def push(self, val):
        self.__list.append(val)

    def pop(self):
        return self.__list.pop()

    def __iter__(self):
        return iter(self.__list)

stack = Stack()
stack.push(1)
stack.push(2)
for s in stack:
    print(s)
```

Example

```
$ python stack.py
1
2
```

Sorting

Python list provides a built-in `list.sort` method which sorts a list in-place without using extra memory. Moreover, the return value of `list.sort` is `None` in order to avoid confusion with `sorted` and the function can only be used for `list`.

```
>>> l = [5, 4, 3, 2, 1]
>>> l.sort()
>>> l
[1, 2, 3, 4, 5]
>>> l.sort(reverse=True)
>>> l
[5, 4, 3, 2, 1]
```

The `sorted` function does not modify any iterable object in-place. Instead, it returns a new sorted list. Using `sorted` is safer than `list.sort` if some list's elements are read-only or immutable. Besides, another difference between `list.sort` and `sorted` is that `sorted` accepts any **iterable object**.

```
>>> l = [5, 4, 3, 2, 1]
>>> new = sorted(l)
>>> new
[1, 2, 3, 4, 5]
>>> l
[5, 4, 3, 2, 1]
>>> d = {3: 'andy', 2: 'david', 1: 'amy'}
>>> sorted(d)  # sort iterable
[1, 2, 3]
```

To sort a list with its elements are tuples, using `operator.itemgetter` is helpful because it assigns a key function to the `sorted` key parameter. Note that the key should be comparable; otherwise, it will raise a `TypeError`.

```
>>> from operator import itemgetter
>>> l = [('andy', 10), ('david', 8), ('amy', 3)]
>>> l.sort(key=itemgetter(1))
>>> l
[('amy', 3), ('david', 8), ('andy', 10)]
```

`operator.itemgetter` is useful because the function returns a getter method which can be applied to other objects with a method `__getitem__`. For example, sorting a list with its elements are dictionary can be achieved by using `operator.itemgetter` due to all elements have `__getitem__`.

```
>>> from pprint import pprint
>>> from operator import itemgetter
>>> l = [
...     {'name': 'andy', 'age': 10},
...     {'name': 'david', 'age': 8},
...     {'name': 'amy', 'age': 3},
... ]
>>> l.sort(key=itemgetter('age'))
>>> pprint(l)
[{'age': 3, 'name': 'amy'},
 {'age': 8, 'name': 'david'},
 {'age': 10, 'name': 'andy'}]
```

If it is necessary to sort a list with its elements are neither comparable nor having `__getitem__` method, assigning a customized key function is feasible.

```
>>> class Node(object):
...     def __init__(self, val):
...         self.val = val
...     def __repr__(self):
...         return f"Node({self.val})"
...
>>> nodes = [Node(3), Node(2), Node(1)]
>>> nodes.sort(key=lambda x: x.val)
>>> nodes
[Node(1), Node(2), Node(3)]
>>> nodes.sort(key=lambda x: x.val, reverse=True)
>>> nodes
[Node(3), Node(2), Node(1)]
```

The above snippet can be simplified by using `operator.attrgetter`. The function returns an attribute getter based on the attribute's name. Note that the attribute should be comparable; otherwise, `sorted` or `list.sort` will raise `TypeError`.

```
>>> from operator import attrgetter
>>> class Node(object):
...     def __init__(self, val):
...         self.val = val
...     def __repr__(self):
...         return f"Node({self.val})"
...
>>> nodes = [Node(3), Node(2), Node(1)]
>>> nodes.sort(key=attrgetter('val'))
>>> nodes
[Node(1), Node(2), Node(3)]
```

If an object has `__lt__` method, it means that the object is comparable and `sorted` or `list.sort` is not necessary to input a key function to its key parameter. A list or an iterable sequence can be sorted directly.

```
>>> class Node(object):
...     def __init__(self, val):
...         self.val = val
...     def __repr__(self):
...         return f"Node({self.val})"
...     def __lt__(self, other):
...         return self.val - other.val < 0
...
>>> nodes = [Node(3), Node(2), Node(1)]
>>> nodes.sort()
>>> nodes
[Node(1), Node(2), Node(3)]
```

If an object does not have `__lt__` method, it is likely to patch the method after a declaration of the object's class. In other words, after the patching, the object becomes comparable.

```
>>> class Node(object):
...     def __init__(self, val):
...         self.val = val
...     def __repr__(self):
...         return f"Node({self.val})"
...
>>> Node.__lt__ = lambda s, o: s.val < o.val
>>> nodes = [Node(3), Node(2), Node(1)]
>>> nodes.sort()
>>> nodes
[Node(1), Node(2), Node(3)]
```

Note that `sorted` or `list.sort` in Python3 does not support `cmp` parameter which is an **ONLY** valid argument in Python2. If it is necessary to use an old comparison function, e.g., some legacy code, `functools.cmp_to_key` is useful since it converts a comparison function to a key function.

```
>>> from functools import cmp_to_key
>>> class Node(object):
...     def __init__(self, val):
...         self.val = val
...     def __repr__(self):
...         return f"Node({self.val})"
...
>>> nodes = [Node(3), Node(2), Node(1)]
>>> nodes.sort(key=cmp_to_key(lambda x,y: x.val - y.val))
>>> nodes
[Node(1), Node(2), Node(3)]
```