The functionality of a data structure can be represented as a collection of methods, each with given parameters and a given result from calling it. For example, the Python list has the methods `append`, `count` and `index` that add an element to the list, count the number of occurrences of an element and search for the index of an element.

By defining a class we can create a customized data structure that has its own set of methods. Often such a data structure contains some standard Python data structure such as a list or a dictionary as internal data storage. The advantage of a class is that it offers a clean interface that hides the details of the internal implementation.

Example: Stack

> **Task**

> Implement a class `Stack` that implements the stack data structure. The class should have the following methods:
>
> - `push(x)` : add the element `x` to the top of the stack
> - `top()` : access the element at the top of the stack
> - `pop()` : remove the element at the top of the stack
>
> The time complexity of each method should be $O(1)$.

The class `Stack` is easy to implement using the Python list, which supports additions and removals at the end of the list in $O(1)$ time. We can implement the class as follows:

```
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, x):
        self.stack.append(x)
```

```
    def top(self):
        return self.stack[-1]

    def pop(self):
        self.stack.pop()
```

The idea is that the class defines an internal list `stack` that stores the elements of the stack. The methods `push` and `pop` are implemented using the list methods `append` and `pop`, and the method `top` is implemented by accessing the last element of the list.

The following code tests the functionality of the class:

```
s = Stack()
s.push(1)
s.push(2)
s.push(3)
print(s.top()) # 3
print(s.top()) # 3
s.pop()
print(s.top()) # 2
```

Using the class `Stack` instead of using the list directly restricts the functionality to the stack methods. For example, access is allowed only to the last element of the list. The user of the class does not need to know the internal implementation, but can trust that the methods `push`, `top` and `pop` are available.

Notice that we can access the internal data too if we know how the class is implemented. This is illustrated by the following code:

```
s = Stack()
s.push(1)
s.push(2)
s.push(3)
print(s.stack) # [1, 2, 3]
```

Here the `print` command accesses all elements of the internal list, even though only the last element is accessible through the methods of the class `Stack`.

How not to implement a class

The following way of implementing a class does not work:

```
class Stack:
    stack = []

    def push(self, x):
        self.stack.append(x)

    def top(self):
        return self.stack[-1]

    def pop(self):
        self.stack.pop()
```

The difference to the earlier class is that now there is no constructor (method `__init__`) and the list `stack` is created at the main level of the class. This can appear to work at first try:

```
s = Stack()
s.push(1)
s.push(2)
s.push(3)
print(s.top()) # 3
```

The trouble is that the list `stack` is *shared* by all objects created from the class. This is illustrated in the following code:

```
a = Stack()
b = Stack()
a.push(1)
b.push(2)
print(a.top()) # 2
```

The code adds the number `1` to the stack `a` and the number `2` to the stack `b`. Then the code accesses the top element of the stack `a`. The result should be `1` but here it is `2` because of the shared list `stack`. Adding an element to one stack adds it to both stacks, which is not how the class should function.

Additional class features

The earlier class `Stack` is correct but is still missing some useful features. First, the printout of a stack is not very informative:

```
s = Stack()
s.push(1)
s.push(2)
s.push(3)
print(s) # <__main__.Stack object at 0x7fd6c4e23ee0>
```

We can address this by adding a method `__repr__` that produces a description of the contents of the stack as a string. In this case, the method can turn the list `stack` into a string and return that:

```
    def __repr__(self):
        return str(self.stack)
```

After this change, we can print out the contents of a stack:

```
s = Stack()
s.push(1)
s.push(2)
s.push(3)
print(s) # [1, 2, 3]
```

Another feature missing is that we cannot determine the size of the stack with the function `len`. Trying to do that would cause an error. We can fix this by defining the method `__len__`:

```
    def __len__(self):
        return len(self.stack)
```

The method `__len__` is called when an object of the class is given as parameter to the function `len`. With our class, the method can return the length of the internal list `stack`. Now the following code works as expected:

```
s = Stack()
s.push(1)
s.push(2)
s.push(3)
print(len(s)) # 3
```

There is still a potential problem with the methods `pop` and `top`. What happens if the list is empty when they are called? In such a situation, there is no element to remove or access. We can add error handling to the methods to cover such situations as follows:

```
    def pop(self):
        if len(self.stack) == 0:
            raise IndexError("pop from empty stack")
        return self.stack.pop()

    def top(self):
        if len(self.stack) == 0:
            raise IndexError("top from empty stack")
        return self.stack[-1]
```

Now, if the methods are called when the stack is empty, they produce the error `IndexError` with a text describing the cause of the error. This is illustrated by the following code:

```
s = Stack()
s.push(1)
s.pop()
s.pop() # IndexError: pop from empty stack
```

Example: Efficient duplicates

> **Task**

> Implement a class `SuperStack` with the following methods:
>
> - `push(x)`: add the element `x` to the top of the stack
> - `push_many(k, x)`: add `k` copies of the element `x` to the top of the stack

- `top()` : access the element at the top of the stack
- `pop()` : remove the element at the top of the stack

The time complexity of each method should be $O(1)$.

This class `SuperStack` is the same as the earlier class `Stack` but with the addition of the method `push_many`, which adds multiple copies of the same element to the stack. Implementing the new method is a challenge, because it should run in $O(1)$ time independent of how many copies of the element is added.

The following straightforward implementation is not efficient enough:

```
    def push_many(self, x, k):
        for i in range(k):
            self.push(x)
```

Here the loop needs $O(k)$ time and its running time depends on the number of copies added. To achieve the time complexity $O(1)$, the method should have no loop.

A useful approach to class design is to separate what is required and how it is achieved. Here the class is required to have the four methods with time complexity $O(1)$, but we are free to choose the internal implementation. Specifically, there is no requirement to store each element separately in a stack as long as the methods work *as if* there was such a stack.

An efficient way to implement the class is to store a stack of pairs of the form \((k,x)\), which represents $k\($ copies of the element\)x$$. For example, the code

```
s = SuperStack()
s.push_many(3, 8)
s.push(4)
s.push_many(2, 5)
```

creates the stack \([(3,8),(1,4),(2,5)]\) as a compact representation of the stack \([8,8,8,4,5,5]\). Now each call to the methods `push` or `push_many` adds just one element to the compact stack. The methods `top` and `pop` have to be modified to correctly process the pairs in the stack. The class can be implemented as follows:

```
class SuperStack:
    def __init__(self):
        self.stack = []

    def push(self, x):
        self.stack.append((1, x))

    def push_many(self, k, x):
        self.stack.append((k, x))

    def top(self):
        return self.stack[-1][1]

    def pop(self):
        last = self.stack[-1]
        if last[0] == 1:
            self.stack.pop()
        else
            self.stack[-1] = (last[0] - 1, last[1])
```

The method `top` returns the second element of the top pair, since that is the actual element in the stack. The method `pop` looks at the first element, the repeat count, of the top pair. If the repeat count is one, the whole pair is removed. Otherwise, the repeat count is reduced by one.

With this implementation, each method of the class runs in \(O(1)\) time, since now there are no loops.

Example: Mode

> **Task**

> Implement a class `Mode` with the following methods:
>
> - `add(x)` : add the number `x` on the list
> - `mode()` : return the mode of the list, i.e., the most frequent number on the list
>
> The time complexity of each method should be \(O(1)\).

This is similar to the task of computing the mode of a given list that we saw in an earlier chapter. The difference is that now we can alternate additions to the list

with queries for the mode among the current list contents. For example, the class could be used as follows:

```
m = Mode()
m.add(1)
m.add(1)
m.add(2)
print(m.mode()) # 1
m.add(2)
m.add(2)
print(m.mode()) # 2
```

Since the mode query is the only query that the class needs to answer, we do not need to keep the full list but just enough information to be able to compute the mode. We can use a dictionary to store the occurrence count of each number. In addition, the class keeps track of the current mode.

```
class Mode:
    def __init__(self):
        self.count = {}
        self.status = (0, 0)

    def add(self, x):
        if x not in self.count:
            self.count[x] = 0
        self.count[x] += 1
        self.status = max(self.status, (self.count[x], x))

    def mode(self):
        return self.status[1]
```

The dictionary `count` contains the occurrence counts of the elements, and the pair `status` stores the current mode in the form \((k,x)\): the mode is \(x\) and it occurs \(k\) times on the list. When an element \(x\) is added to the list, the mode is updated if \(x\) is the mode after the addition. A succinct way to do this is to use the function `max`, which uses the first element of the pair (the count) as the primary key in the comparison.

Since neither method of the class has loops or slow operations, their time complexity is \(O(1)\) as required.

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI