1 Fundamental datastructures and abstract data types

You are already acquainted with some basic datastructures in Python:

- Tuples
- Lists
- Dictionaries

These are very handy and quite easy to use since Python is a high-level language.

They can be described in terms of more “classic” datastructures:

- Tuples – immutable arrays, fixed size.
- Lists – arrays, adjustable size, with interfaces implementing the behaviour of stacks and queues.
- Dictionaries – generalized hash-tables.

We will look more closely at hash tables in a few weeks.

The basic datastructures are only good when they fit in RAM memory. If you have extreme needs (truly Big Data!), then you might need other implementations (look at modules) or implement datastructures yourself. Ideally, with same/similar interface!

1.1 The array

Fixed size vector. Python’s builtin list supercedes the array.

- Random element access

1.2 The stack

Linear structure with Last-In, First-Out (LIFO) functionality.

1.2.1 Formal description

Not quite like in Cormen et al!

Use an array $S$ of size $n$, let $top(S)$ point to the first empty element. If $top(S) = 0$ then the stack is empty.

Stack-Empty($S$)

if $top(S) = 0$
then return True
else return False

Push($S$, $x$)

$S[top(S)] := x$
$top(S) := top(S) + 1$

Pop($S$)

if Stack-Empty($S$) then
error "Underflow"
else
    top(S) := top(S) - 1
    return S[top(S)]

Note: three operations are described are part the interface to a stack. What more is needed?

- What is needed to make an actual Python class for a stack?
- How should we think about n?

1.3 The queue

The queue has First-In, First-Out functionality. Two basic operations: enqueue and dequeue.

Use an array Q with n elements again. head(Q) points to the first element in the queue, and tail(Q) is the first empty position in the queue.

From Cormen et al:

Enqueue(Q, x)
    Q[tail(Q)] := x
    if tail(Q) = n then
        tail(Q) := 1
    else
        tail(Q) := tail(Q) + 1

Dequeue(Q)
    x := Q[head(Q)]
    if head(Q) = length(Q) then
        head(Q) := 1
    else
        head(Q) := head(Q) + 1
    return x

What is needed to make this into an actual Python class?

1.4 Linked lists

The linked lists are classic data structures for linear access that Python has “retired”. There are two basic forms: single-link and double-link. Draw illustrations.

The built-in Python list, which behaves as an array, stack, or queue, supercedes the lists. In languages like C++, Java, etc, the linked lists are still important, but typically implemented in a standard library. In C, you typically still implement lists yourself.

Some characteristics:

- In the list’s basic forms, you don’t keep track of the length.
- You don’t have random access. To get to element i, you have to traverse i elements of the list.
• It is cheap to remove any element. Python lists are array-based, so requires moving lots of elements. You might not notice that when coding, but you will notice it if working on large lists/arrays!

1.5 Abstract datatypes and concrete datatypes

A concrete datatype is the actual implementation of a datatype. In Python, much of the concreteness is abstracted away, and you don’t see it. This is not the case in lower-level programming languages like C. In modern C++, Java, and many other languages let you get away with abstractions when you need to and support concretisation when you so desire.

A key to good programming is to use abstractions. For data, that is using datastructures that manages the data through well-defined interfaces (gränssnitt). The interface is an agreement, proposed by an implementer, for how to use the datastructure. A user (programmer) should neither need to, nor be able to, use implementation details for the datastructure. The implementer is free to change internal details of the datastructure, but promises to keep the interface as long as possible.

There is a conflict between concrete and abstract datatypes. On one hand, you should strive for general and reusable abstractions. On the other, you might want, or need, space or time efficient datastructures, requiring a lot of thought and engineering in the concrete details. However, it is possible to have simple, elegant, and efficient datastructures (one can argue that Python’s dictionaries is an example).

1.5.1 Pythonic abstractions

One reason these datastructures are so easy to use in Python is that they have well-defined and general interfaces, making them easy to use. There are also two “hidden” interface making them easy to put into pythonic idioms. First, the tuple, list, string, and dictionary datatypes are examples of sequence objects. This makes it possible to write \( x[i] \) regardless of what \( x \) is. Slicing, concatenation etc also follow due to a common sequence interface. Neither the types nor the shared interfaces are unique, but Python does it well.

The second notable interface is the iterator interface and is used, for example, in the in notation \( (x \text{ in } s) \). You can adapt this interface by implementing the iterator protocol (two methods):

- \texttt{__iter__}(): Return an iterator object, can be from a altogether different class.
- \texttt{__next__}(): Implemented in the iterator object. Return the next item in the datastructure. When there are no further items, raise the \texttt{StopIteration} exception.

The iterator object can be \texttt{self}, if there is no reason to implement a special iterator class. When can that be useful/necessary?

- If you want independent iterators simultaneously.
- Several different iterators needed. E.g., trees – you might want iterators for leaves, internal nodes, and all nodes.
- Complicated implementation can be good to put separately.

The advantage with adapting this interface in your classes is that you get a lot of pythonic behaviour for free. This leads to
• pretty code,
• expressive code, and
• good readability.

Side note: read about iterators at

• https://docs.python.org/3/tutorial/classes.html#iterators
• http://anandology.com/python-practice-book/iterators.html (examples for 2.7!)

1.5.2 Fibonacci example

Here is a simple way of computing Fibonacci numbers:

```python
def fibonacci(n):
    """
    Return the largest Fibonacci number smaller than n
    """
    a = 0
    b = 1
    while b < n:
        a, b = b, a+b
    return a
```

This is from *Dive into Python 3* is an example of how to turn the `fibonacci` function into an
class, abstracting the computation using the iterator interface. Note: the class is not equivalent
to the function.

class Fib:
    """iterator that yields numbers in the Fibonacci sequence"""
    def __init__(self, max):
        self.max = max

    def __iter__(self):
        self.a = 0
        self.b = 1
        return self

    def __next__(self):
        fib = self.a
        if fib > self.max:
            raise StopIteration
        self.a, self.b = self.b, self.a + self.b
        return fib

Usage:

```python
fib_obj = Fib(20)
for fibs in fib_obj:
    print(fibs)
```