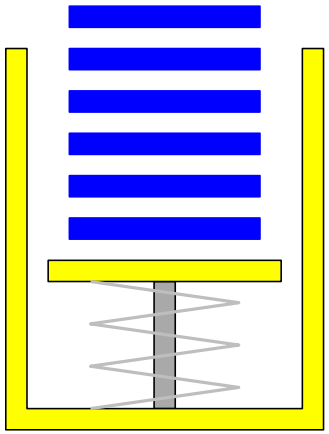
Abstract Data Type Stack

Think about a stack of books, or dishes.

One can only access the top of the stack.



Operations on a stack:

- push something on top,
- look at the top,
- pop something off the stack.

Reversing a string

A simple stack client:

```
def reverse(s):
  S = Stack()
  for ch in s:
    S.push(ch)
  while not S.is_empty():
    ch = S.pop()
    print(ch, end="")
  print()
```


Balanced symbol checker

`(){[(){}](())}` is correct,
but `(){[({)}](())}` is not correct.

How to check whether a string is balanced:

1. Make an empty stack,
2. For each symbol in the string:
   (a) If the symbol is an opening symbol, push it on the stack.
   (b) If it is a closing symbol, then
      i. If the stack is empty, return false.
      ii. If the top of the stack does not match the closing symbol, return false.
      iii. Pop the stack.
3. Return true if the stack is empty, otherwise false.

The runtime stack

Where do variables live? The local variables of a method live inside the method's activation record (also called stack frame).

(But all objects are on the heap. The stack frame only stores the variable names and references to the heap.)

When a method starts executing, its stack frame is created. When the method returns, its stack frame is destroyed.

The Python runtime system keeps stack frames on a stack.

The top of the stack is the stack from of the currently executing method. A stack is suitable for storing stack frames, since the start and return time of methods form a nesting structure (like balanced parentheses).

(The runtime stack is built into the Python interpreter! It is not a Python object—we cannot access the stack of activation records ourselves.)

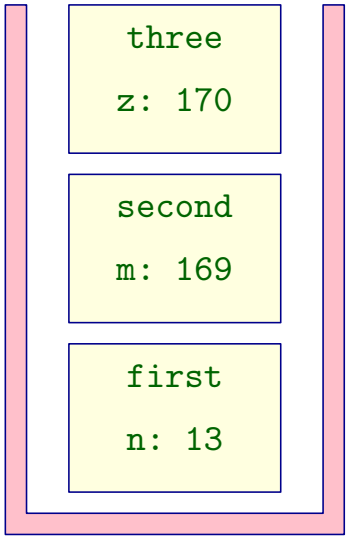
Example

```
def first(n):
  second(n)
  second(n * n)

def second(m):
  three(m)
  three(m+1)
  three(m+2)

def three(z):
  print("In three(%d):" % z)

first(13)
```



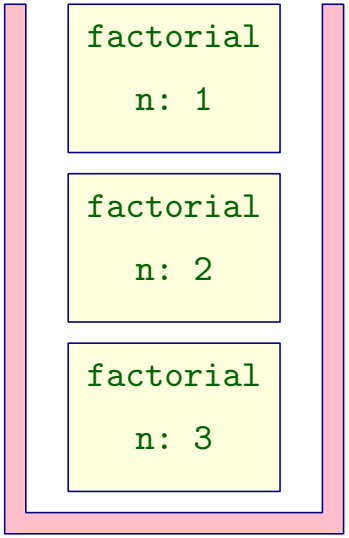
example1.py

Recursion

The runtime stack makes recursion possible.

```
def factorial(n):
  if n <= 1:
    return 1
  else:
    return n * factorial(n - 1)
```

Any recursive program can be rewritten to use a stack and no recursion.



example2.py

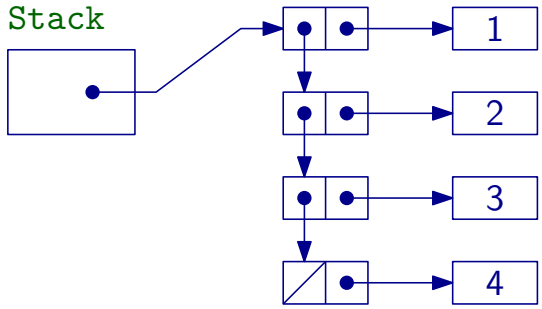
Implementing a stack

- As a Python List:
  `is_empty` and `top` take constant time, `push` and `pop` take constant time on average (because sometimes the internal array needs to be made larger or smaller).
- If we want every operation to be always fast, we need to know the maximum stack size in advance. Then we can make an array of the maximum size and keep an index to the top of the stack.
- If we do not know the maximum size, but we need every `push` and `pop` operation to be fast, then we cannot use a list.

A stack without lists

We use a small `Node` object to hold each stack element.

The `Stack` holds a reference to the `Node` at the top of the stack.
Each `Node` holds a reference to the one directly below in the stack.



Now each operation can be done in constant time!