Stacks

Stack is an abstract data type that stores a collection of elements, and behaves like a stack of books, or a stack of serving plates in the cafeteria: you can only access the top element.

Formally, a stack supports the following operations:

- `push(el)` pushes an element onto the stack. It becomes the new top element.
- `top()` returns the element at the top of the stack.
- `pop()` returns the element at the top of the stack and removes it from the stack. The element just below it becomes the new top element.
- `is_empty()` returns true if there are no elements on the stack.

When the stack is empty, `top()` and `pop()` will raise an exception.

A stack is sometimes called a LIFO, since it works on the principle of *Last In First Out*.

A simple client

As an example of client code that exploits the LIFO behavior of stacks, here is a function that prints a string in reverse order:

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

Checking balanced parentheses

Let's write a function that checks whether an arithmetic expression is correctly balanced. This is much simpler than fully parsing the expression—in fact, we on only need to look at the different kinds of parentheses in the expression, and can ignore all other symbols.

For instance, `(){[(){}]([])}` is correctly balanced, but `(){[({)}]([])}` is not (because a round `)` closes a curly `{` bracket).

We use the following strategy:

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.

You can find the code in `balanced.py`.

Stack implementation

Using a Python list. A simple stack implementation is given in the `liststack` module. It uses a Python list to store the stack elements. `push` performs a list append, `pop` uses a list `pop()`, removing the last element from the list.

All operations work in constant time on average, as we discussed earlier. But some `push()` operations take longer, since the array used internally by the Python list is full, and a new array needs to be allocated—such a `push()` operation takes time linear in the current size of the stack.

With given maximum size. For a Python program, it's unlikely that you need a stronger guarantee. But if you a writing `C` code running on an embedded circuit, you may need to guarantee that a function returns within a few microseconds. In such an application, we need a stack that guarantees constant time for each operation.

This is easy to achieve if we know the maximum size that the stack will ever have in advance. We simply create a list of sufficient size, and maintain the index of the current top of stack ourselves.

You can find the implementation in the `arraystack` module.

A linked stack. If we do not know the maximum stack size in advance, then we simply cannot use a Python list or array—we need to work with smaller objects.

The solution is to create small objects called *nodes*. Each node stores (a reference to) one element of the stack, and also a reference to the *next* node. The `Stack` object itself only knows about the node at the top of the stack. This node has a reference to the node just below the top of the stack, and so on. The last node stores `None` instead of a reference to a following node.

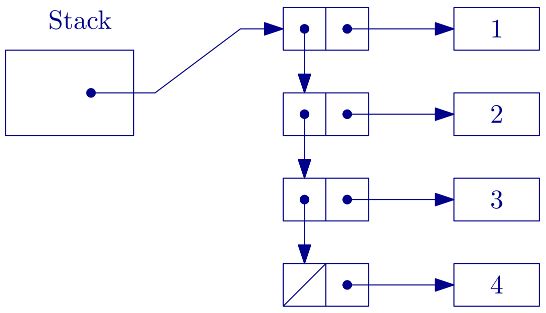

Figure 1: A stack implemented using small nodes.

You can find the implementation in the `linkedstack` module.

Stack frames

We learnt earlier that the local variables of a function are stored in its *stack frame*. This is a block of memory that is created every time the function is called, and which is destroyed when the function returns.

Perhaps you already guessed it: the Python interpreter keeps all stack frames on a *stack*. When a function is called, its stack frame is created and pushed onto the stack. When the function returns, its stack frame is popped from the stack and destroyed. Therefore, the currently executing function is always at the top of the stack. (Observe that when a function `f` returns, the new top of the stack must be the stack frame of the function that had called `f`.)

Why does a stack work for storing stack frames? Because the start times and return times of functions form a nicely nested structure, just like the balanced parentheses. You cannot return from a function `f` unless all the functions that were called after the current call to `f` have already returned.

The runtime stack that stores the stack frames is built into the Python interpreter. It is not a Python object itself.

When we discussed recursion, you may have wondered how recursion really works. How does the Python interpreter distinguish between all these calls to a recursive function `f`? Doesn't it get confused?

The answer is: it's the runtime stack makes recursion possible. Each call to a recursive function `f` has its own stack frame, with its own copy of the local variables, and the most recent call is always at the top of the stack. This also explains why there is limit to the recursion depth: by default, the Python runtime stack is limited to 1000 stack frames.

One consequence of this insight is the following: you can always rewrite a recursive function in non-recursive form using a stack. Essentially, you use the stack to "simulate" the runtime stack.