If the stack is nonempty we return its top-most value and a `nil` error value. Since Go uses 0-based indexing the first element in a slice or array is at position 0 and the last element is at position `len(`*`sliceOrArray`*`) - 1`.

There is no formality when returning more than one value from a function or method; we simply list the types we are returning after the function or method's name and ensure that we have at least one `return` statement that has a corresponding list of values.

```
func (stack *Stack) Pop() (interface{}, error) {
    theStack := *stack
    if len(theStack) == 0 {
        return nil, errors.New("can't Pop() an empty stack")
    }
    x := theStack[len(theStack)-1]  ❶
    *stack = theStack[:len(theStack)-1]  ❷
    return x, nil
}
```

The `Stack.Pop()` method is used to remove and return the top (last added) item from the stack. Like the `Stack.Top()` method it returns the item and a `nil` error, or if the stack is empty, a `nil` item and a non-`nil` error.

The method must have a receiver that is a pointer since it modifies the stack by removing the returned item. For syntactic convenience, rather than ~~refer~~ to `*stack` (the actual stack that the `stack` variable points to) throughout the method, we assign the actual stack to a local variable (`theStack`), and work with that variable instead. This is quite cheap because `*stack` is pointing to a `Stack`, which uses a slice for its representation, so we are really assigning little more than a reference to a slice.

If the stack is empty we return a suitable error. Otherwise we retrieve the stack's top (last) item and store it in a local variable (`x`). Then we take a slice of the stack (which itself is a slice). The new slice has one less element than the original and is immediately set to be the value that the `stack` pointer points to. And at the end, we return the retrieved value and a `nil` error. We can reasonably expect any decent Go compiler to reuse the slice, simply reducing the slice's length by one, while leaving its capacity unchanged, rather than copying all the data to a new slice.

The item to return is retrieved (~~28~~ ❶) using the `[]` index operator with a single index; in this case the index of the slice's last element.

The new slice is obtained by using the `[]` slice operator with an index range (~~28~~ ❷). An index range has the form *`first`*`:`*`end`*. If *`first`* is omitted—as here—0 is assumed, and if *`end`* is omitted, the `len()` of the slice is assumed. The slice thus obtained has elements with indexes from and including the *`first`* up to and

```
func π(places int) *big.Int {
    digits := big.NewInt(int64(places))
    unity := big.NewInt(0)
    ten := big.NewInt(10)
    exponent := big.NewInt(0)
    unity.Exp(ten, exponent.Add(digits, ten), nil)  ❶
    pi := big.NewInt(4)
    left := arccot(big.NewInt(5), unity)
    left.Mul(left, big.NewInt(4))  ❷
    right := arccot(big.NewInt(239), unity)
    left.Sub(left, right)
    pi.Mul(pi, left)  ❸
    return pi.Div(pi, big.NewInt(0).Exp(ten, ten, nil))  ❹
}
```

The π() function begins by computing a value for the unity variable ($10^{digits+10}$) which we use as a scale factor so that we can do all our calculations using integers. The +10 adds an extra ten digits to those given by the user, to avoid rounding errors. We then use Machin's formula with our modified arccot() function (not shown) that takes the unity variable as its second argument. Finally, we return the result divided by 10^{10} to reverse the effects of the unity scale factor.

To get the unity variable to hold the correct value we begin by creating four variables, all of type *big.Int (i.e., pointer to big.Int; see §4.1. ~~Values, Pointers, and Reference Types~~ ➤ 138). The unity and exponent variables are initialized to 0, the ten variable to 10, and the digits variable to the number of digits requested by the user. The unity computation is performed in a single line (~~63 ➤~~, ❶). The big.Int.Add() method adds 10 to the number of digits. Then the big.Int.Exp() method is used to raise 10 to the power of its second argument (*digits* + 10). When used with a nil third argument—as here—big.Int.Exp(*x*, *y*, nil), performs the computation x^y; with three non-nil arguments, big.Int.Exp(*x*, *y*, *z*), computes $x^y \bmod z$. Notice that we did not need to assign to unity; this is because most big.Int methods modify their receiver as well as return it, so here, unity is modified to have the resultant value.

The rest of the computation follows a similar pattern. We set an initial value of pi to 4 and then compute the inner left-hand part of Machin's formula. Again notice that we don't need to assign to left after creating it (~~63 ➤~~, ❷), since the big.Int.Mul() method stores the result in its receiver (i.e., in this case in variable left) as well as returning the result (which we can safely ignore). Next we compute the inner right-hand part of the formula and subtract the right from the left (leaving the result in left). Now we multiply pi (of value 4) by left (which holds the result of Machin's formula). This produces the result but scaled by unity. So in the final line (~~63 ➤~~, ❹) we reverse the scaling by dividing the result (in pi) by 10^{10}.

```
xs := []int{2, 4, 6, 8}
fmt.Println("5 @", Index(xs, 5), "  6 @", Index(xs, 6))
ys := []string{"C", "B", "K", "A"}
fmt.Println("Z @", Index(ys, "Z"), "  A @", Index(ys, "A"))
```

```
5 @ -1   6 @ 2
Z @ -1   A @ 3
```

What we really need to be able to do is treat the slice generically—that way we could have just one loop and do the type-specific testing inside it. Here is a function that achieves this—and it produces the same output as the above code snippet if we replace calls to `Index()` with calls to `IndexReflectX()`.

```
func IndexReflectX(xs interface{}, x interface{}) int { // Long-winded way
    if slice := reflect.ValueOf(xs); slice.Kind() == reflect.Slice {
        for i := 0; i < slice.Len(); i++ {
            switch y := slice.Index(i).Interface().(type) {
            case int:
                if y == x.(int) {
                    return i
                }
            case string:
                if y == x.(string) {
                    return i
                }
            }
        }
    }
    return -1
}
```

The function begins by using Go's reflection support (provided by the `reflect` package; §9.4.9, ➤ 425), to convert the `xs interface{}` into a slice-typed `reflect.Value`. Such values provide the methods we need to traverse the slice's items and to extract any items we are interested in. Here, we access each item in turn and use the `reflect.Value.Interface()` function to pull out the value as an `interface{}` which we immediately assign to `y` inside a type `switch`. This ensures that `y` has the item's actual type (e.g., `int` or `string`) which can be directly compared with the unchecked type-asserted `x` value.

In fact, the `reflect` package can take on far more of the work, so we can ~~considerably~~ simplify this function.

```
func IndexReflect(xs interface{}, x interface{}) int {
    if slice := reflect.ValueOf(xs); slice.Kind() == reflect.Slice {
```