solving the problem she faces. As long as the plurality of tools does not get into the way, she can choose among them as best fits the situation. This has been expressed very elegantly by Bjarne Stroustrup in *The Design and Evolution of C++* (1994, p. 23):

> My interest in computers and programming languages is fundamentally pragmatic.
>
> I feel most at home with the empiricists rather than with the idealists.... That is, I tend to prefer Aristotle to Plato, Hume to Descartes, and shake my head sadly over Pascal. I find comprehensive "systems" like those of Plato and Kant fascinating, yet fundamentally unsatisfying in that they appear to me dangerously remote from everyday experiences and the essential peculiarities of individuals.
>
> I find Kierkegaard's almost fanatical concern for the individual and keen psychological insights much more appealing than the grandiose schemes and concern for humanity in the abstract of Hegel or Marx. Respect for groups that doesn't include respect for individuals of those groups isn't respect at all. Many C++ design decisions have their roots in my dislike for forcing people to do things in some particular way. In history, some of the worst disasters have been caused by idealists trying to force people into "doing what is good for them." Such idealism not only leads to suffering among its innocent victims, but also to delusion and corruption of the idealists applying the force. I also find idealists prone to ignore experience and experiment that inconveniently clashes with dogma or theory. Where ideals clash and sometimes even when pundits seem to agree, I prefer to provide support that gives the programmer a choice.

Going back to Smalltalk, consider the problem of going over a collection of objects, applying a function on each one of them, and collecting the results. This is implemented as follows:

```
collect: aBlock
    "Evaluate aBlock with each of the receiver's elements as the argument.
    Collect the resulting values into a collection like the receiver. Answer
    the new collection."

    | newCollection |
    newCollection := self species new.
    self do: [:each | newCollection add: (aBlock value: each)].
    ^ newCollection
```

To understand this method, it is enough to know that the `species` method returns either the class of the receiver or a class similar to it—the difference is too subtle to make a difference to us here. What is interesting is that to construct the new collection we only need something that has a selector called `value`, which we call. Blocks do have a selector called `value`, so any block can be used. But the fact that we are talking about blocks is incidental: anything that implements `value` will do.

Anything that returns a value has turned out to be important enough in programming that it has earned a name: it is now called a *function object*, and is a fundamental constituent of C++ STL algorithms. Traditionally, it is a staple of functional programming, usually called a *map* function. It is of course available in Lisp. It is also offered in Python, allowing us to do things like: