Chapter 3
Programs and Data

Object-oriented programming is a popular way of organizing programs, by grouping together data with the procedures that operate on them, and by facilitating some kinds of modularity and abstraction. In the context of our PCAP framework, object-oriented programming will give us methods for capturing common patterns in data and the procedures that operate on that data, via classes, generic functions, and inheritance.

In this chapter, we will try to develop a deep understanding of object-oriented programming by working through the mechanism by which an interpreter evaluates a computer program, starting from the simplest single-statement programs and working up through list structures, procedures, objects, and classes. Although we use Python as an example, this discussion is intended to be illustrative of principles of computer languages, more generally.

In many computer languages, including Python, programs are understood and executed by a computer program called an interpreter. Interpreters are surprisingly simple: the rules defining the meaning or semantics of a programming language are typically short and compact. The enormous richness and complexity of computer programs comes from the composition of primitive elements with simple rules. We will study the meaning of computer programs by understanding how the interpreter operates on them.

Programs should never be a mystery to you: you can learn the simple semantic rules and, if necessary, simulate what the interpreter would do, in order to understand any computer program you are faced with.

3.1 Primitives, Composition, Abstraction, and Patterns

Let’s start by thinking about how the PCAP framework applies to computer programs, in general, and filling in table 3.1. We’ll explore the PCAP ideas in data, procedures, and objects.

Data
The primitive data items in most programming languages are things like integers, floating point numbers, and strings. We can combine these into data structures (we discuss some basic Python data structures in section ??) such as lists, arrays, and records. Making a data structure allows us, at the most basic level, to think of a collection of primitive data elements as if it were one thing, freeing us from details. Sometimes, we just want to think of a collection of data, not in terms of its underlying representation, but in terms of what it represents. So, we might want to think
of a set of objects, or a family tree, without worrying whether it is an array or a list in its basic representation. Abstract data types provide a way of abstracting away from representational details to think about what the data really means.

**Procedures**

The primitive procedures of a language are things like built-in numeric operations and basic list operations. We can combine these with the facilities of the language, such as if and while, or function composition (f(g(x))). If we want to abstract away from the details of how a particular computation is done, we can define a new function; defining a function allows us to use it for computational jobs without thinking about the details of how those computational jobs get done. One way to capture common patterns of abstraction in procedures is to abstract over procedures themselves, with higher-order procedures, which we discuss in detail in section ??.

**Objects**

Object-oriented programming provides a number of methods of abstraction and pattern capture in both data and procedures. At the most basic level, objects can be used as records, combining together primitive data elements. More generally, they provide strategies for jointly abstracting a data representation and the procedures that work on it. The features of inheritance and polymorphism are particularly important, and we will discuss them in detail later in this chapter.

<table>
<thead>
<tr>
<th></th>
<th>Procedures</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Primitives</strong></td>
<td>+, *, ==</td>
<td>numbers, strings</td>
</tr>
<tr>
<td><strong>Means of combination</strong></td>
<td>if, while, f(g(x))</td>
<td>lists, dictionaries, objects</td>
</tr>
<tr>
<td><strong>Means of abstraction</strong></td>
<td>def</td>
<td>ADTS, classes</td>
</tr>
<tr>
<td><strong>Means of capturing patterns</strong></td>
<td>higher-order procedures</td>
<td>generic functions, inheritance</td>
</tr>
</tbody>
</table>

Table 3.1 Primitives, combination, abstraction, patterns framework for computer programs

### 3.2 Expressions and assignment

We can think of most computer programs as performing some sort of transformation on data. Our program might take as input the exam scores of everyone in the class and generate the average score as output. Or, in a transducer model, we can think about writing the program that takes the current memory state of the transducer and an input, and computes a new memory state and output.

To represent data in a computer, we have to encode it, ultimately as sequences of binary digits (0s and 1s). The memory of a computer is divided into ’words’, which typically hold 32 or 64 bits;
a word can be used to store a number, one or several characters, or a pointer to (the address of) another memory location.

A computer program, at the lowest level, is a set of primitive instructions, also encoded into bits and stored in the words of the computer’s memory. These instructions specify operations to be performed on the data (and sometimes the program itself) that are stored in the computer’s memory. In this class, we won’t work at the level of these low-level instructions: a high-level programming level such as Python lets us abstract away from these details. But it’s important to have an abstract mental model of what is going on in the computer.

3.2.1 Simple expressions

A cornerstone of a programming language is the ability to evaluate expressions. We’ll start here with arithmetic expressions, just to get the idea. An expression consists of a sequence of ‘tokens’ (words, special symbols, or numerals) that represent the application of operators to data elements. Each expression has a value, which can be computed recursively by evaluating primitive expressions, and then using standard rules to combine their values to get new values.

Numerals, such as 6 or -3.7 are primitive expressions, whose values are numeric constants. Their values can be integers, within some fixed range dictated by the programming language, or floating point numbers. Floating point numbers are used to represent non-integer values, but they are different, in many important ways, from the real numbers. There are infinitely many real numbers within a finite interval, but only finitely many floating-point numbers exist at all (because they all must be representable in a fixed number of bits). In fact, the usual laws of real arithmetic (transitivity, associativity, etc.) are violated in floating-point arithmetic, because the results of any given sub-computation may not be representable in the given number of bits.

We will illustrate the evaluation of expressions in Python by showing short transcripts of interactive sessions with the Python shell: the shell is a computer program that

- Prompts the user for an expression, by typing `>>>`,
- Reads what the user types in,
- Evaluates the expression, and
- Prints out the resulting value

So, for example, we might have this interaction with Python:

```
>>> 2 + 3
5
>>> (3 * 8) - 2
22
>>> 2.0
2.0
>>> 0.1
0.10000000000000001
>>> 1.0 / 3.0
0.33333333333333331
```
There are a couple of things to observe here. First, we can see how floating point numbers only approximately represent real numbers: when we type in \(0.1\), the closest Python can come to it in floating point is \(0.10000000000000001\). The last interaction is particularly troubling: it seems like the value of the expression \(1 / 3\) should be something like \(0.33333\). However, in Python, if both operands to the \(/\) operator are integers, then it will perform an integer division, truncating any remainder.\(^9\)

These expressions can be arbitrarily deeply nested combinations of primitives. The rules used for evaluation are essentially the same as the ones you learned in school; the interpreter proceeds by applying the operations in precedence order,\(^10\) evaluating sub-expressions to get new values, and then evaluating the expressions those values participate in, until a single value results.

### 3.2.2 Variables

We can’t go very far without variables. A variable is a name that we can bind to have a particular value and then later use in an expression. When a variable is encountered in an expression, it is evaluated by looking to see what value it is bound to.

An interpreter keeps track of which variables are bound to what values in binding environments. An environment specifies a mapping between variable names and values. The values can be integers, floating-point numbers, characters, or pointers to more complex entities such as procedures or larger collections of data.

Here is an example binding environment:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(b)</td>
<td>(3)</td>
</tr>
<tr>
<td>(x)</td>
<td>(2.2)</td>
</tr>
<tr>
<td>(foo)</td>
<td>(-1012)</td>
</tr>
</tbody>
</table>

Each row represents a binding: the entry in the first column is the variable name and the entry in the second column is the value it to which it is bound.

When you start up the Python shell, you immediately start interacting with a local binding environment. You can add a binding or change an existing binding by evaluating an assignment statement of the form:

\[
<\text{var}> = <\text{expr}>
\]

---

\(^9\) This behavior will no longer be the default in Python 3.0.

\(^10\) Please Excuse My Dear Aunt Sally (Parentheses, Exponentiation, Multiplication, Division, Addition, Subtraction)
where `<var>` is a variable name (a string of letters or digits or the character `_`, not starting with a digit) and `<expr>` is a Python expression.\(^\text{11}\)

We might have the following interaction in a fresh Python shell:

```python
>>> a = 3
>>> a
3
>>> b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
>>> 
```

We started by assigning the variable `a` to have the value 3. That added a binding for `a` to the local environment.

Next, we evaluated the expression `a`. The value of an expression with one or more variable names in it cannot be determined unless we know what environment it is being evaluated with respect to. Thus, we will always speak of evaluating expressions in an environment. During the process of evaluating an expression in some environment `E`, if the interpreter comes to a variable, it looks that variable up in `E`: if `E` contains a binding for the variable, then the associated value is returned; if it does not, then an error is generated. In the Python shell interaction above, we can see that the interpreter was able to find a binding for `a` and return a value, but it was not able to find a binding for `b`.

Why do we bother defining values for variables? They allow us to re-use an intermediate value in a computation. We might want to compute a formula in two steps, as in:

```python
>>> c = 952**4
>>> c**2 + c / 2.0
6.7467650588636822e+23
```

They will also play a crucial role in abstraction and the definition of procedures.

It is fine to reassign the value of a variable; although we use the equality symbol `=` to stand for assignment, we are not making a mathematical statement of equality. So, for example, we can write:

```python
>>> a = 3
>>> a = a + 1
>>> a
4
```

\(^{11}\) When we want to talk about the abstract form or syntax of a programming language construct, we will often use meta-variables, written with angle brackets, like `<var>`. This is meant to signify that `<var>` could be any Python variable name, for example.
Exercise 3.1. What is the result of evaluating this sequence of assignment statements and the last expression? Determine this by hand-simulating the Python interpreter. Draw an environment and update the stored values as you work through this example.

```python
>>> a = 3
>>> b = a
>>> a = 4
>>> b
```

3.3 Structured data

We will often want to work with large collections of data. Rather than giving each number its own name, we want to organize the data into natural structures: grocery lists, matrices, sets of employee medical records. In this section, we will explore a simple but enormously useful and flexible data structure, which is conveniently built into Python: the list. The precise details of how lists are represented inside a computer vary from language to language. We will adopt an abstract model in which we think of a list as an ordered sequence of memory locations that contain values. So, for example, in Python, we can express a list of three integers as:

```python
>>> [1, 7, -2]
[1, 7, -2]
```

which we’ll draw in an abstract memory diagram as:

```
1 7 -2
```

We can assign a list to a variable:

```python
>>> a = [2, 4, 9]
```

A binding environment associates a name with a single fixed-size data item. So, if we want to associate a name with a complex structure, we associate the name directly with a ‘pointer’ to (actually, the memory address of) the structure. So we can think of `a` as being bound to a ‘pointer’ to the list:

```
a | 2 4 9
```
Now that we have lists, we have some new kinds of expressions, which let us extract components of a list by specifying their indices. An index of 0 corresponds to the first element of a list. An index of -1 corresponds to the last element (no matter how many elements there are). So, if a is bound as above, then we’d have:

```python
>>> a[0]
2
>>> a[2]
9
>>> a[-1]
9
>>> a[3]
```

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

Note that if we attempt to access an element of the list that is not present (in this case, the fourth element of a three-element list), then an error is generated.

Lists can be nested inside one another. The Python expression:

```python
>>> c = [3, [1], [2, 1], [[4]]]
```

creates a list that looks, in memory, like this:

![Diagram of nested lists]

It’s also possible to have an empty list. It’s written in Python as []. We’ll draw it in our memory diagrams as a small black box. So, for example, this list

```python
>>> z = [3, [], [[]]]
```

looks like this in memory:

![Diagram of nested lists with empty list]

---

12 See the Python tutorial for much more on list indexing.
Python has a useful function, `len`, which takes a list as an argument and returns its length. It does not look inside the elements of the list—it just returns the number of elements at the top level of structure. So, we have

```python
>>> len([1, 2, 3])
3
>>> len([[1, 2, 3]])
1
```

**Exercise 3.2.** Draw a diagram of the binding environment and memory structure after the following statement has been evaluated:

```python
a = [[1], 2, [3, 4]]
```

**Exercise 3.3.** Draw a diagram of the binding environment and memory structure after the following statement has been evaluated:

```python
a = [[[]]]
```
Exercise 3.4. Give a Python statement which, when evaluated, would give rise to this memory structure:

What is the value, in this environment, of the following expressions:

- c[1]
- c[-1]
- c[2][1]

3.3.1 List mutation and shared structure

Lists are *mutable* data structures, which means that we can actually change the values stored in their elements. We do this by using element-selection expressions, like a[1] on the left-hand side of an assignment statement. So, the assignment

\[ a[1] = -3 \]

assigns the second element of \( a \) to be \(-3\). If that statement were evaluated in this environment,

then the resulting environment would be:

We have permanently changed the list named \( a \).
In this section, we will explore the consequences of the mutability of lists; programs that change list structure can become very confusing, but you can always work your way through what is happening by drawing out the memory diagrams.

Continuing the previous example, let’s remember that `a` is bound directly to a pointer to a list, and think about what happens if we do:

```python
>>> b = a
```

Now, `a` and `b` are both names for the same list structure, resulting in a memory diagram like this:

![Memory Diagram](image)

Now, we can reference parts of the list through `b`, and even change the list structure that way:

```python
>>> b[0]
2
>>> b[2] = 1
```

Notice that, because `a` and `b` point to the same list, changing `b` changes `a`!

```python
>>> a
[2, -3, 1]
```

Here’s the memory picture now:

![Memory Diagram](image)

Another important way to change a list is to add or delete elements. We’ll demonstrate adding elements to the end, but see the Python documentation for more operations on lists. This statement

```python
>>> a.append(9)
```

causes a new element to be added to the end of the list name `a`. The resulting memory state is:

![Memory Diagram](image)
As before, because a and b are names for the same list, b is changed too:

```python
>>> b
[2, -3, 1, 9]
```

Often, it will be important to make a fresh copy of a list so that you can change it without affecting the original one. Here are two equivalent ways to make a copy (use whichever one you can remember):

```python
>>> c = list(a)
>>> c = a[:]
```

Here is a picture of the memory at this point:

![Memory diagram](image)

Now, if we change an element of c, it doesn’t affect a (or b!):

```python
>>> c[0] = 100
>>> c
[100, -3, 1, 9]
>>> a
[2, -3, 1, 9]
```

We can make crazy lists that share structure within a single list:

```python
>>> f = [1, 2, 3]
>>> g = [1, f, [f]]
>>> g
[1, [1, 2, 3], [[1, 2, 3]]]
```

which results in this memory structure:

![Memory diagram](image)

If you want to add an element to a list and get a new copy at the same time, you can do
The `+` operator makes a new list that contains the elements of both of its arguments, but doesn’t share any top-level structure. All of our methods of copying only work reliably if your lists don’t contain other lists, because it only copies one level of list structure. So, for example, if we did:

```python
>>> h = list(g)
```

we would end up with this picture:

It’s clear that if we were to change `f`, it would change `h`, so this isn’t a completely new copy. If you need to copy deep structures, you will need to use the Python `copy.deepcopy` procedure.

**Exercise 3.5.** Give a sequence of Python statements which, when evaluated, would give rise to this memory structure:

![Memory Structure Diagram 1](image)

**Exercise 3.6.** Give a sequence of Python statements which, when evaluated, would give rise to this memory structure:

![Memory Structure Diagram 2](image)
Exercise 3.7. Show the memory structure after this sequence of expressions.

```python
global a
 password = [1, 2, 3]
 global b
 b = [a, a]
 a.append(100)
```

What will be the value of b at this point?

Exercise 3.8. Show the memory structure after this sequence of expressions.

```python
global a
global b
global c
 a = [5, 6]
b = [1, 2]
c = b + a
```

We’ll use this “curvy road” symbol to indicate sections of the notes or exercises that are somewhat more difficult and not crucial to understanding the rest of the notes. Feel free to skip them on first reading; but, of course, we think they’re cool.\(^\text{13}\)

Exercise 3.9.

Show the memory structure after this sequence of expressions.

```python
>>> a = [1, 2, 3]
>>> a[1] = a
```

What will be the value of a at this point?

### 3.3.2 Tuples and strings

Python has two more list-like data types that are important to know about.

A tuple is a structure that is like a list, but is not mutable. You can make new tuples, but you cannot change the contents of a tuple or add elements to it. A tuple is typically written like a list, but with round parentheses instead of square ones:

```python
>>> a = (1, 2, 3)
```

In fact, it is the commas and not the parentheses that matter here. So, you can write

---

\(^{13}\) Thanks to Don Knuth’s Art of Computer Programming for the idea of the curvy road sign.
and still get a tuple. The only tricky thing about tuples is making a tuple with a single element. We could try

```
>>> a = (1)
>>> a
1
```

but it doesn’t work, because in the expression (1) the parentheses are playing the standard grouping role (and, in fact, because parentheses don’t make tuples). So, to make a tuple with a single element, we have to use a comma:

```
>>> a = 1,
>>> a
(1,)
```

This is a little inelegant, but so it goes.

Tuples will be important in contexts where we are using structured objects as ‘keys’, that is, to index into another data structure, and where inconsistencies would occur if those keys could be changed.

An important special kind of tuple is a string. A string can almost be thought of as a tuple of characters. The details of what constitutes a character and how they are encoded is complicated, because modern character sets include characters from nearly all the world’s languages. We’ll stick to the characters we can type easily on our keyboards. In Python, you can write a string with either single or double quotes: ‘abc’ or "abc". You can select parts of it as you would a list:

```
>>> s = 'abc'
>>> s[0]
'a'
>>> s[-1]
'c'
```

The strange thing about this is that s is a string, and because Python has no special data type to represent a single character, s[0] is also a string.

We will frequently use + to concatenate two existing strings to make a new one:

```
>>> to = 'Jody'
>>> fromP = 'Robin'
>>> letter = 'Dear ' + to + '
It's over.
' + fromP
>>> print letter
Dear Jody,
It’s over.
Robin
```
As well as using + to concatenate strings, this example illustrates several other small but important points:

- You can put a single quote inside a string that is delimited by double-quote characters (and vice versa).
- If you want a new line in your string, you can write \n. Or, if you delimit your string with a triple quote, it can go over multiple lines.
- The print statement can be used to print out results in your program.
- Python, like most other programming languages, has some reserved words that have special meaning and cannot be used as variables. In this case, we wanted to use from, but that has a special meaning to Python, so we used fromP instead.

**Structured assignment**

Once we have lists and tuples, we can use a nice trick in assignment statements, based on the packing and unpacking of tuples.

```python
>>> a, b, c = 1, 2, 3
>>> a
1
>>> b
2
>>> c
3
```

Or, with lists,

```python
>>> [a, b, c] = [1, 2, 3]
>>> a
1
>>> b
2
>>> c
3
```

When you have a list (or a tuple) on the left-hand side of an assignment statement, you have to have a list (or tuple) of matching structure on the right-hand side. Then Python will “unpack” them both, and assign to the individual components of the structure on the left hand side. You can get fancier with this method:

```python
>>> thing = [8, 9, [1, 2], 'John', [33.3, 44.4]]
>>> [a, b, c, d, [e1, e2]] = thing
>>> c
[1, 2]
>>> e1
33.299999999999997
```
3.4 Procedures

Procedures are computer-program constructs that let us:

• Gather together sequences of statements
• Abstract away from particular data items they operate on

Here is a procedure definition,\(^{14}\) and then its use:

```python
def square(x):
    return x * x
```

```plaintext
>>> square(6)
36
>>> square(2 - square(2))
4
```

We’ll work through, in detail, what happens when the interpreter evaluates a procedure definition, and then the application of that procedure.

3.4.1 Definition

A procedure definition has the abstract form:

```python
def <name>(<fp1>, …, <fpn>):
    <statement1>
    …
    <statementk>
```

There are essentially three parts:

• `<name>` is a name for the procedure, with the same restrictions as a variable name;
• `<fp1>, …, <fpn>` is a list of formal parameters, which will stand for the data items this procedure will operate on; and
• `<statement1>, …, <statementk>`, known as the body of the procedure, is a list of Python statements (right now, we know about assignment statements, print statements, and basic expressions, but there will be more.)

When we evaluate a procedure definition in an environment, E, Python does two things:

---

\(^{14}\)In the code displayed in these notes, we will show procedures being defined and then used, as if the definitions were happening in the Python shell (but without the prompts). In fact, you shouldn’t type procedure definitions into the shell, because if you make a mistake, you’ll have to re-type the whole thing, and because multi-line objects aren’t handled very well. Instead, type your procedure definitions into a file in Idle, and then test them by ‘running’ the file in Idle (which will actually evaluate all of the expressions in your file) and then evaluating test expressions in Idle’s shell.
1. Makes a procedure object that contains the formal parameters, the body of the procedure, and a pointer to $E$; and then
2. Binds the name to have this procedure as its value.

### 3.4.2 Procedure calls

When you evaluate an expression of the form

```plaintext
<expr0>(<expr1>, ..., <exprn>)
```

the Python interpreter treats this as a procedure call. It will be easier to talk about a specific case of calling a procedure, so we will illustrate with the example

```plaintext
>>> square(a + 3)
```

evaluated in this environment ($E_1$):

Here are the steps:

1. The expression that determines the procedure ($<expr0>$) is evaluated. In this case, we evaluate `square` in $E_1$ and get Procedure1.
2. The expressions that determine the arguments ($<expr1>, \ldots, <exprn>$) are evaluated. In this case, we evaluate `a + 3` in $E_1$ and get 5.
3. A new environment (in this case $E_2$) is created, which:
   - binds the formal parameters of the procedure (in this case $x$) to the values of its arguments (in this case, 5); and
   - has as its parent the environment in which the procedure was defined (we find a pointer to this environment stored in the procedure; in this case $E_1$ is its parent).

At this point, our memory looks like this:

---

15 In our memory diagrams, we will show the procedure object as a box with the word `Procedure<N>`, where $N$ is some integer, at the top; we give the procedure objects these numbers so we can refer to them easily in the text.
Procedure1
(x)
return x*x

E1
x 5

E2
The dotted line between E2 and E1 is intended to indicate that E1 is the parent environment of E2.

4. The statements of the procedure body are evaluated in the new environment until either a return statement or the end of the list of statements is reached. If a return statement is evaluated, the expression after the return is evaluated and its value is returned as the value of the procedure-call expression. Otherwise, the procedure has no return value, and the expression has the special Python value None.

In our example, the only statement in the body is a return statement. So, we evaluate the expression \( x \times x \) in E2, obtaining a value of 25. That value is returned as the value of the entire procedure-call expression \( \text{square}(a + 3) \).

This basic mechanism can generate behavior of arbitrary complexity, when coupled with recursion or other control structures, discussed in section 2.3.

Worked example 1

Let’s examine what happens when we evaluate the following Python code:

```python
def p(x, y):
    z = x*x - y
    return z + 1/z

>>> p(1.0, 2.0)
-2.0
```

Here is a picture of the calling environment (E1) and the procedure-call environment (E2) just before the body of the procedure is evaluated:

After evaluating \( z = x\times y \) in E2, we have:
Finally, we evaluate \( z + \frac{1}{z} \) in E2 and get -2.0, which we return.

**Worked example 2**

Here’s another example:

```python
def silly(a):
    a = a + 1
    return a
```

>>> b = 6
>>> silly(b)
7

Here is a picture of the calling environment (E1) and the procedure-call environment (E2) just before the body of the procedure is evaluated:

After evaluating \( a = a + 1 \) in E2, we have:

Finally, we evaluate \( a \) in E2 and get 7, which we return.

Convince yourself that the execution below works fine, and notice that having a binding for \( a \) in E2 means that we leave the binding for \( a \) in E1 unmolested:
Exercise 3.10. Draw the environment diagram at the time the statement with the arrow is being executed:

```python
def fizz(x, y):
    p = x + y
    q = p*p
    return q + x  # ------------
```

```python
>>> fizz(2, 3)
```

Exercise 3.11. Draw the environment diagram at the time the statement with the arrow is being executed:

```python
def fuzz(a, b):
    return a + b  # ------------

def buzz(a):
    return fuzz(a, square(a))

>>> buzz(2)
```

3.4.3 Non-local references

So far, whenever we needed to evaluate a variable, there was a binding for that variable in the 'local' environment (the environment in which we were evaluating the expression). But consider this example:

```python
def biz(a):
    return a + b

>>> b = 6
>>> biz(2)
```

This actually works fine, and returns 8. Let’s see why. Here is the environment, E1, that the Python shell is working in, after we execute the `b = 6` statement:
Now, when we evaluate \texttt{biz(2)} in \texttt{E1}, we make a new environment, \texttt{E2}, which binds \texttt{a} to 2 and points to \texttt{E1} as its parent environment.

We need to elaborate, slightly, how it is that a variable \( v \) is evaluated in an environment \( E \):

- We look to see if there is a binding for \( v \) in \( E \); if so, we stop and return it.
- If not, we evaluate \( v \) in the parent environment of \( E \). If \( E \) has no parent, we generate an error.

It is important to appreciate that this process will continue up an arbitrarily long chain of environments and their parents until either a binding for \( v \) is found or there are no more environments to look in.

So, in our case, we will evaluate the expression \( a + b \) in environment \texttt{E2}. We start by evaluating \( a \) and finding value 2 in \texttt{E2}. Then, we evaluate \( b \) and can’t find it in \texttt{E2}...but we don’t panic! We follow the parent pointer to \texttt{E1} and try again. We find a binding for \( b \) in \texttt{E1} and get the value 6. So, the value of \( a + b \) in \texttt{E2} is 8.

**Exercise 3.12.** Draw a picture of the relevant binding environments when the statement with the arrow is being executed. What value does the procedure return?

```python
def f(a):
    return a + g(a)

def g(b):
    return a + b  # <----------

>>> f(3)
```
Exercise 3.13. Draw a picture of the relevant binding environments when the statement with the arrow is being executed. What value does the procedure return?

```python
def f(a):
def g(b):
    return a + b  # <----------
    return a + g(a)

>>> f(3)
```

Exercise 3.14. Draw a picture of the relevant binding environments when the statement with the arrow is being executed. What value does the procedure return? Does it cause an error?

```python
def a(a):
    return a * a

>>> a(2)
```

### 3.4.4 Environments in Python

Generally, Python establishes the following binding environments:

1. `__builtin__`: the mother of all environments: it contains the definitions of all sorts of basic symbols, like `list` and `sum`. It is the parent of all module environments.

2. Module: each separate file that contains Python code is called a module and establishes its own environment, whose parent is `__builtin__`.

3. Procedure calls: as described in this section. A procedure that is defined at the ‘top level’ of a module (that is, not nested in the definition of another procedure) has the module’s environment as its parent, and has its name defined in the module’s environment. Procedures that are defined inside other procedures have the procedure-call environment of the containing procedure as their parent.

We have seen two operations that cause bindings to be created: assignments and procedure calls. Bindings are also created when you evaluate an `import` statement. If you evaluate

```python
import math
```

then a file associated with the `math` module is evaluated and the name `math` is bound, in the current environment, to the `math` module, which is an environment. No other names are added.
to the current environment, and if you want to refer to names in that module, you have to qualify them, as in `math.sqrt`. If you execute

```python
from math import sqrt
```

then the `math` file is evaluated, and the name `sqrt` is bound, in the current environment, to whatever the name `sqrt` is bound to in the `math` module. But note that if you do this, the name `math` isn’t bound to anything, and you can’t access any other procedures in the `math` module unless you import them explicitly, as well.

### 3.4.5 Non-local references in procedures

There is an important subtlety in the way names are handled in the environment created by a procedure call. When a name that is not bound in the local environment is referred to, then it is looked up in the chain of parent environments. So, as we’ve seen, it is fine to have

```python
a = 2
def b():
    return a
```

When a name is assigned inside a procedure, a new binding is created for it in the environment associated with the current call of that procedure. So, it is fine to have

```python
a = 2
def b():
a = a + 1
c = 4
return a + c
```

Both assignments cause new bindings to be made in the local environment, and it is those bindings that are used to supply values in the return expression. It will not change `a` in the global environment.

But here is a code fragment that causes trouble:

```python
a = 3
def b():
a = a + 1
print a
```

It seems completely reasonable, and you might expect `b()` to return 4. But, instead, it generates an error. What’s going on?? It all has to do with when Python decides to add a binding to the local environment. When it sees this procedure definition, it sees that the name `a` occurs on the left-hand-side of an assignment statement, and so, at the very beginning, it puts a new entry for `a` in the local environment, but without any value bound to it. Now, when it’s time to evaluate the statement
a = a + 1

Python starts by evaluating the expression on the right hand side: a + 1. When it tries to look up the name a in the procedure-call environment, it finds that a has been added to the environment, but hasn’t yet had a value specified. So it generates an error.

In Python, we can write code to increment a number named in the global environment, by using the global declaration:

```python
a = 3
def b():
    global a
    a = a + 1
    print a
```

```plaintext
>>> b()
4
>>> b()
5
>>> a
5
```

The statement global a asks that a new binding for a not be made in the procedure-call environment. Now, all references to a are to the binding in the module’s environment, and so this procedure actually changes a.

In Python, we can only make assignments to names in the procedure-call environment or to the module environment, but not to names in intermediate environments. So, for example,

```python
def outer():
    def inner():
        a = a + 1
    a = 0
    inner()
```

In this example, we get an error, because Python has made a new binding for a in the environment for the call to inner. We’d really like for inner to be able to see and modify the a that belongs to the environment for outer, but there’s no way to arrange this. Some other programming languages, such as Scheme, offer more fine-grained control over how the scoping of variables is handled.

### 3.4.6 Section ??Procedures as first-class objects

In Python, unlike many other languages, procedures are treated in much the same way as numbers: they can be stored as values of variables, can be passed as arguments to procedures, and can be returned as results of procedure calls. Because of this, we say that procedures are treated as “first-class” objects. We will explore this treatment of “higher-order” procedures (procedures that manipulated procedures) throughout this section.
First of all, it’s useful to see that we can construct (some) procedures without naming them using the \texttt{lambda} constructor:

\texttt{lambda <var1>, \ldots, <varn> : <expr>}

The formal parameters are <var1>, \ldots, <varn> and the body is <expr>. There is no need for an explicit \texttt{return} statement; the value of the expression is always returned. A single expression can only be one line of Python, such as you could put on the right hand side of an assignment statement. Here are some examples:

\begin{verbatim}
>>> f = lambda x: x*x
>>> f
<function <lambda> at 0x4ecf0>
>>> f(4)
16
\end{verbatim}

Here is a procedure of two arguments defined with \texttt{lambda}:

\begin{verbatim}
>>> g = lambda x,y : x * y
>>> g(3, 4)
12
\end{verbatim}

Using the expression-evaluation rules we have already established, we can do some fancy things with procedures, which we’ll illustrate throughout the rest of this section.

\begin{verbatim}
>>> procs = [lambda x: x, lambda x: x + 1, lambda x: x + 2]
>>> procs[0]
<function <lambda> at 0x83d70>
>>> procs[1](6)
7
\end{verbatim}

Here, we have defined a list of three procedures. We can see that an individual element of the list (e.g., \texttt{procs[0]}) is a procedure.\footnote{In the programming world, people often use the words “function” and “procedure” either interchangeable or with minor subtle distinctions. The Python interpreter refers to the objects we are calling procedures as “functions.”} So, then \texttt{procs[1]}(6) applies the second procedure in the list to the argument 6. Since the second procedure returns its argument plus 1, the result is 7.

Here is a demonstration that procedures can be assigned to names in just the way other values can.

\begin{verbatim}
>>> fuzz = procs[2]
>>> fuzz(3)
5

def thing(a):
    return a * a
\end{verbatim}
Passing procedures as arguments

Just as we can pass numbers or lists into procedures as arguments, we can pass in procedures as arguments, as well.

What if we find that we’re often wanting to perform the same procedure twice on an argument? That is, we seem to keep writing `square(square(x))`. If it were always the same procedure we were applying twice, we could just write a new procedure

```
def squaretwice(x):
    return square(square(x))
```

But what if it’s different procedures? We can write a new procedure that takes a procedure \( f \) as an argument and applies it twice:

```
def doTwice(f, x):
    return f(f(x))
```

So, if we wanted to square twice, we could do:

```
>>> doTwice(square, 2)
16
```

Here is a picture of the environment structure in place when we are evaluating the `return f(f(x))` statement in `doTwice`:

Environment \( E_1 \) is the module environment, where procedures `square` and `doTwice` were defined and where the expression `doTwice(square, 2)` is being evaluated. The interpreter:

- Evaluates `doTwice` in \( E_1 \) and gets `Procedure6`.
- Evaluates `square` in \( E_1 \) and gets `Procedure5`. 
• Evaluates 2 in E1 and gets 2.
• Makes the new binding environment E2, binding the formal parameters, f and x, of Procedure 6, to actual arguments Procedure5 and 2.
• Evaluates the body of Procedure6 in E2.

Now, let’s peek in one level deeper to the process. To evaluate f(f(x)) in E2, the interpreter starts by evaluating the inner f(x) expression. It
• Evaluates f in E2 and gets Procedure5.
• Evaluates x in E2 and gets 2.
• Makes the new binding environment E3, binding the formal parameter x, of Procedure5, to 2.
• Evaluates the body of Procedure5 in E3, getting 4.

The environments at the time of this last evaluation step are:

A similar thing happens when we evaluate the outer application of f, but now with argument 4, and a return value of 16.
Exercise 3.15. Here is the definition of a procedure `sumOfProcs` that takes two procedures, `f` and `g`, as well as another value `x`, as arguments, and returns `f(x) + g(x)`. The `sumOfProcs` procedure is then applied to two little test procedures:

```python
def sumOfProcs(f, g, x):
    return f(x) + g(x)

def thing1(a):
    return a*a*a

def thing2(b):
    return b+1  # ---------------

>>> sumOfProcs(thing1, thing2, 2)
```

Draw a picture of all of the relevant environments at the moment the statement with the arrow is being evaluated. What is the return value of the call to `sumOfProcs`?

Returning procedures as values

Another way to apply a procedure multiple times is this:

```python
def doTwiceMaker(f):
    return lambda x: f(f(x))
```

This is a procedure that returns a procedure! If you’d rather not use `lambda`, you could write it this way:

```python
def doTwiceMaker(f):
    def twoF(x):
        return f(f(x))
    return twoF
```

Now, to use `doTwiceMaker`, we might start by calling it with a procedure, such as `square`, as an argument and naming the resulting procedure.

```python
>>> twoSquare = doTwiceMaker(square)
```

Here is a picture of the environments just before the `return twoF` statement in `doTwiceMaker` is evaluated.
Here is a picture of the environments after the doTwiceMaker returns its value and it is assigned to twoSquare in E1. It’s important to see, here, that Procedure8 is the return value of the call to doTwiceMaker and that, because Procedure8 retains a pointer to the environment in which it was defined, we need to keep E2 around. And it is E2 that remembers what procedure (via its binding for f) that is going to be applied twice.

Now, when we evaluate this expression in E1

```
>>> twoSquare(2)
```

we start by making a new binding environment, E3, for the procedure call. Note that, because the procedure we’re calling, Procedure8, has E2 stored in it, we set the parent of E3 to be E2.
Next, we evaluate the body of Procedure8, which is \( \text{return } f(f(x)) \) in E3. Let’s just consider evaluating the inner expression \( f(x) \) in E3. We evaluate \( f \) in E3 and get Procedure5, and evaluate \( x \) and get 2. Now, we make a new binding environment, E4, to bind the formal parameter of Procedure5 to 2. Because Procedure5 has a stored pointer to E1, E4’s parent is E1, as shown here:

Evaluating the body of Procedure5 in E4 yields 4. We will repeat this process to evaluate the outer application of \( f \), in \( f(f(x)) \), now with argument 4, and end with result 16.

Essentially the same process would happen when we evaluate

```python
>>> doTwiceMaker(square)(2)
16
```

except the procedure that is created by the expression \( \text{doTwiceMaker(square)} \) is not assigned a name; it is simply used as an intermediate result in the expression evaluation.
3.5 Recursion

There are many control structures in Python, and other modern languages, which allow you write short programs that do a lot of work. In this section, we discuss recursion, which is also a way to write programs of arbitrary complexity. It is of particular importance here, because the structure of a language interpreter is recursive.

We have seen how we can define a procedure, and then can use it without remembering or caring about the details of how it is implemented. We sometimes say that we can treat it as a black box, meaning that it is unnecessary to look inside it to use it. This is crucial for maintaining sanity when building complex pieces of software. An even more interesting case is when we can think of the procedure that we’re in the middle of defining as a black box. That’s what we do when we write a recursive procedure.

Recursive procedures are ways of doing a lot of work. The amount of work to be done is controlled by one or more arguments to the procedure. The way we are going to do a lot of work is by calling the procedure, over and over again, from inside itself! The way we make sure this process actually terminates is by being sure that the argument that controls how much work we do gets smaller every time we call the procedure again. The argument might be a number that counts down to zero, or a string or list that gets shorter.

There are two parts to writing a recursive procedure: the base case(s) and the recursive case. The base case happens when the thing that’s controlling how much work you do has gotten to its smallest value; usually this is 0 or the empty string or list, but it can be anything, as long as you know it’s sure to happen. In the base case, you just compute the answer directly (no more calls to the recursive procedure!) and return it. Otherwise, you’re in the recursive case. In the recursive case, you try to be as lazy as possible, and foist most of the work off on another call to this procedure, but with one of its arguments getting smaller. Then, when you get the answer back from the recursive call, you do some additional work and return the result.

Here’s an example recursive procedure that returns a string of n 1’s:

```python
def bunchaOnes(n):
    if n == 0:
        return ''
    else:
        return bunchaOnes(n-1) + '1'
```

The thing that’s getting smaller is n. In the base case, we just return the empty string. In the recursive case, we get someone else to figure out the answer to the question of n-1 ones, and then we just do a little additional work (adding one more ’1’ to the end of the string) and return it.

Exercise 3.16. What is the result of evaluating

```
bunchaOnes(-5)
```
Here’s another example. It’s kind of a crazy way to do multiplication, but logicians love it.

```python
def mult(a,b):
    if a==0:
        return 0
    else:
        return b + mult(a-1,b)
```

Trace through an example of what happens when you call `mult(3, 4)`, by adding a print statement that prints arguments `a` and `b` as the first line of the procedure, and seeing what happens.

Here’s a more interesting example of recursion. Imagine we wanted to compute the binary representation of an integer. For example, the binary representation of 145 is ‘10010001’. Our procedure will take an integer as input, and return a string of 1’s and 0’s.

```python
def bin(n):
    if n == 0:
        return '0'
    elif n == 1:
        return '1'
    else:
        return bin(n/2) + bin(n%2)
```

The easy cases (base cases) are when we’re down to a 1 or a 0, in which case the answer is obvious. If we don’t have an easy case, we divide up our problem into two that are easier. So, if we convert `n/2` (the integer result of dividing `n` by 2), into a string of digits, we’ll have all but the last digit. And `n%2` (`n` modulo 2) is 1 or 0 depending on whether the number is even or odd, so one more call of `bin` will return a string of ‘0’ or ‘1’. The other thing that’s important to remember is that the `+` operation here is being used for string concatenation, not addition of numbers.

How do we know that this procedure is going to terminate? We know that the number it’s operating on is a positive integer that is getting smaller and smaller, and will eventually be either a 1 or a 0, which can be handled by the base case.

You can also do recursion on lists. Here a way to add up the values of a list of numbers:

```python
def addList(elts):
    if elts == []:
        return 0
    else:
        return elts[0] + addList(elts[1:])
```

The `addList` procedure consumed a list and produced a number. The `incrementElements` procedure below shows how to use recursion to do something to every element of a list and make a new list containing the results.

```python
def incrementElements(elts):
    if elts == []:
        return []
    else:
        return [elts[0]+1] + incrementElements(elts[1:])
```
If the list of elements is empty, then there is no work to be done, and the result is just the empty list. Otherwise, the result is a new list: the first element of the new list is the first element of the old list, plus 1; the rest of the new list is the result of calling incrementElement recursively on the rest of the input list. Because the list we are operating on is getting shorter on every recursive call, we know we will reach the base case, and all will be well.

3.6 Object-oriented programming

We have seen structured data and interesting procedures that can operate on that data. It will often be useful to make a close association between collections of data and the operations that apply to them. The style of programming that takes this point of view is object-oriented programming (OOP). It requires adding some simple mechanisms to our interpreter, but is not a big conceptual departure from the things we’ve already seen. It is, however, a different style of organizing large programs.

3.6.1 Classes and instances

In OOP, we introduce the idea of classes and instances. An instance is a collection of data that describe a single entity in our domain of interest, such as a person or a car or a point in 3D space. If we have many instances that share some data values, or upon which we would want to perform similar operations, then we can represent them as being members of a class and store the shared information once with the class, rather than replicating it in the instances.

Consider the following staff database for a large undergraduate course:

<table>
<thead>
<tr>
<th>name</th>
<th>role</th>
<th>age</th>
<th>building</th>
<th>room</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pat</td>
<td>Prof</td>
<td>60</td>
<td>34</td>
<td>501</td>
<td>6.01</td>
</tr>
<tr>
<td>Kelly</td>
<td>TA</td>
<td>31</td>
<td>34</td>
<td>501</td>
<td>6.01</td>
</tr>
<tr>
<td>Lynn</td>
<td>TA</td>
<td>29</td>
<td>34</td>
<td>501</td>
<td>6.01</td>
</tr>
<tr>
<td>Dana</td>
<td>LA</td>
<td>19</td>
<td>34</td>
<td>501</td>
<td>6.01</td>
</tr>
<tr>
<td>Chris</td>
<td>LA</td>
<td>20</td>
<td>34</td>
<td>501</td>
<td>6.01</td>
</tr>
</tbody>
</table>
There are lots of shared values here, so we might want to define a class. A class definition has the form:

```
class <name>:
    <statement1>
    ...
    <statementn>
```

Here’s the definition of simple class in our example domain:

```
class Staff601:
    course = '6.01'
    building = 34
    room = 501
```

From the implementation perspective, the most important thing to know is that **classes and instances are environments**.

When we define a new class, we make a new environment. In this case, the act of defining class `Staff601` in an environment `E1` results in a binding from `Staff601` to `E2`, an empty environment whose parent is `E1`, the environment in which the class definition was evaluated. Now, the statements inside the class definition are evaluated in the new environment, resulting in a memory state like this:\[17\]

\[\text{Caveat: when we discuss methods in section 3.6.2, we will see that the rules for evaluating procedure definitions inside a class definition are slightly different from those for evaluating procedure definitions that are not embedded in a class definition.}\]

We will often call names that are bound in a class’s environment **attributes** of the class. We can access these attributes of the class after we have defined them, using the *dot* notation:

```
<envExpr>.<var>
```

When the interpreter evaluates such an expression, it first evaluates `<envExpr>`; if the result is not an environment, then an error is signaled. If it is an environment, `E`, then the name `<var>` is looked up in `E` (using the general process of looking in the parent environment if it is not found directly in `E`) and the associated value returned.

So, for example, we could do:

\[\text{In fact, the string ‘6.01’ should be shown as an external memory structure, with a pointer stored in the binding environment; for compactness in the following diagrams we will sometimes show the strings themselves as if they were stored directly in the environment.}\]
We can also use the dot notation on the left-hand side of an assignment statement, if we wish to modify an environment. An assignment statement of the form

\[
\text{<envExpr>}.\text{<var>} = \text{<valExpr>}
\]

causes the name \text{<var>} in the environment that is the result of evaluating \text{<envExpr>} to be bound to the result of evaluating \text{<valExpr>}.

So, we might change the room in which 6.01 meets with:

```python
Staff601.room = Staff601.room - 100
```

or add a new attribute with

```python
Staff601.coolness = 11  # out of 10, of course...
```

Now, we can make an instance of a class with an expression of the form:

\[
\text{<classExpr>()}
\]

This expression has as its value a new empty environment whose parent pointer is the environment obtained as a result of evaluating \text{<classExpr>}.

So, if we do:

```python
>>> pat = Staff601()
```

we will end up with an environment state like this:

```
At this point, given our standard rules of evaluation and the dot notation, we can say:

```python
>>> pat.course
'6.01'
```

The interpreter evaluates \text{pat} in E1 to get the environment E3 and then looks up the name course. It doesn’t find it in E3, so it follows the parent pointer to E2, and finds there that it is bound to '6.01'.
Similarly, we can set attribute values in the instance. So, if we were to do:

```java
pat.name = 'Pat'
pat.age = 60
pat.role = 'Professor'
```

we would get this environment structure.

Note that these names are bound in the instance environment, not the class.

These structures are quite flexible. If we wanted to say that Professor Pat is an exception, holding office hours in a different place from the rest of the 6.01 staff, we could say:

```java
pat.building = 32
pat.office = 'G492'
```

Here is the new environment state. Nothing is changed in the Staff601 class: these assignments just make new bindings in pat’s environment, which ‘shadow’ the bindings in the class, so that when we ask for pat’s room, we get 32.
3.6.2 Methods

Objects and classes are a good way to organize procedures, as well as data. When we define a procedure that is associated with a particular class, we call it a method of that class. Method definition requires only a small variation on our existing evaluation rules.

So, imagine we want to be able to greet 6.01 staff members appropriately on the staff web site. We might add the definition of a salutation method:

```python
class Staff601:
    course = '6.01'
    building = 34
    room = 501

    def salutation(self):
        return self.role + ' ' + self.name
```

This procedure definition, made inside the class definition, is evaluated in *almost* the standard way, resulting in a binding of the name salutation in the class environment to the new procedure. The way in which this process deviates from the standard procedure evaluation process is that the environment stored in the procedure is the the module (file) environment, no matter how deeply nested the class and method definitions are:

Now, for example, we could do:

```python
Staff601.saluation(pat)
```

The interpreter finds that Staff601 is an environment, in which it looks up the name saluation and finds Procedure9. To call that procedure we follow the same steps as in section 3.4.2:
• Evaluate `pat` to get the instance `E3`.
• Make a new environment, `E4`, binding `self` to `E3`. The parent of `E4` is `E1`, because we are evaluating this procedure call in `E1`.
• Evaluate `self.role + ' ' + self.name` in `E4`.
• In `E4`, we look up `self` and get `E3`, look up `role` in `E3` and get 'Professor', etc.
• Ultimately, we return 'Professor Pat'.

Here is a picture of the binding environments while we are evaluating `self.role + ' ' + self.name`.

Note (especially for Java programmers!) that the way the body of `salutation` has access to the attributes of the instance it is operating on and to attributes of the class is via the instance we passed to it as the parameter `self`. The parent environment is `E1`, which means that methods cannot simply use the names of class attributes without accessing them through the instance.

The notation

```python
Staff601.salutation(pat)
```

is a little clumsy; it requires that we remember what class `pat` belongs to, in order to get the appropriate `salutation` method. We ought, instead, to be able to write

```python
pat.salutation(pat)  # Danger: not legal Python
```

This should have exactly the same result. (Verify this for yourself if you don't see it, by tracing through the environment diagrams. Even though the `pat` object has no binding for `saluation`,
the environment-lookup process will proceed to its parent environment and find a binding in the class environment.)

But this is a bit redundant, having to mention pat twice. So, Python added a special rule that says: *If you access a class method by looking in an instance, then that instance is automatically passed in as the first argument of the method.*

So, we can write

```python
pat.salutation()
```

This is *exactly* equivalent to

```python
Staff601.salutation(pat)
```

A consequence of this decision is that every method must have an initial argument that is an instance of the class to which the method belongs. That initial argument is traditionally called `self`, but it is not necessary to do so.

Here is a new method. Imagine that `Staff601` instances have a numeric `salary` attribute. So, we might say

```python
pat.salary = 100000
```

Now, we want to write a method that will give a 6.01 staff member a k-percent raise:

```python
class Staff601:
    course = '6.01'
    building = 34
    room = 501

    def salutation(self):
        return self.role + ' ' + self.name

    def giveRaise(self, percentage):
        self.salary = self.salary + self.salary * percentage

as before, we could call this method as

```python
Staff601.giveRaise(pat, 0.5)
```

or we could use the short-cut notation and write:

```python
pat.giveRaise(0.5)
```

This will change the `salary` attribute of the `pat` instance to 150000.\(^\text{18}\)

---

\(^\text{18}\) Something to watch out for!!! A common debugging error happens when you: make an instance of a class (such as our `pat` above); then change the class definition and re-evaluate the file; then try to test your changes using your old instance, `pat`. Instances remember the definitions of all the methods in their class *when they were created*. So if you change the class definition, you need to make a new instance (it could still be called `pat`) in order to get the new definitions.
3.6.3 Initialization

When we made the `pat` instance, we first made an empty instance, and then added attribute values to it. Repeating this process for every new instance can get tedious, and we might wish to guarantee that every new instance we create has some set of attributes defined. Python has a mechanism to streamline the process of initializing new instances. If we define a class method with the special name `__init__`, Python promises to call that method whenever a new instance of that class is created.

We might add an initialization method to our `Staff601` class:

```python
class Staff601:
    course = '6.01'
    building = 34
    room = 501

    def __init__(self, name, role, years, salary):
        self.name = name
        self.role = role
        self.age = years
        self.salary = salary

    def salutation(self):
        return self.role + ' ' + self.name

    def giveRaise(self, percentage):
        self.salary = self.salary + self.salary * percentage
```

Now, to create an instance, we would do:

```python
pat = Staff601('Pat', 'Professor', 60, 100000)
```

Here is a diagram of the environments when the body of the `__init__` procedure is about to be executed:

---

19 We called the fourth formal parameter `years`, when `age` would have been clearer, just to illustrate that the names of formal parameters do not have to match the attributes they are bound do inside the object.
Chapter 3  Programs and Data

Note that the formal parameter self has been bound to the newly-created instance. Here is the situation after the initialization method has finished executing:

This method seems very formulaic, but it is frequently all we need to do. To see how initialization methods may vary, we might instead do something like this, which sets the salary attribute based on the role and age arguments passed into the initializer.

```python
class Staff601:
    def __init__(self, name, role, age):
        self.name = name
        self.role = role
        if self.role == 'Professor':
            self.salary = 100000
        elif self.role == 'TA':
```

```python
    pass
```
3.6.4 Inheritance

We see that we are differentiating among different groups of 6.01 staff members. We can gain clarity in our code by building that differentiation into our object-oriented representation using subclasses and inheritance.

At the mechanism level, the notion of a subclass is very simple: if we define a class in the following way:

```python
def class <className>(<superclassName):
  <body>
```

then, when the interpreter makes the environment for this new class, it sets the parent pointer of the class environment to be the environment named by `<superclassName>`.

This mechanism allows us to factor class definitions into related, interdependent aspects. For example, in the 6.01 staff case, we might have a base class where aspects that are common to all kinds of staff are stored, and then subclasses for different roles, such as professors:

```python
class Staff601:
  course = '6.01'
  building = 34
  room = 501

def giveRaise(self, percentage):
  self.salary = self.salary + self.salary * percentage

class Prof601(Staff601):
  salary = 100000

  def __init__(self, name, age):
    self.name = name
    self.giveRaise((age - 18) * 0.03)

  def salutation(self):
    return 'Professor' + self.name

Let's trace what happens when we make a new instance of Prof601 with the expression

Prof601('Pat', 60)
```

First a new environment, E4, is constructed, with E3 (the environment associated with the class Prof601 as its parent). Here is the memory picture now:
As soon as it is created, the \_\_init\_ \_ method is called, with the new environment E4 as its first parameter and the rest of its parameters obtained by evaluating the expressions that were passed into to Prof601, in this case, 'Pat' and 60. Now, we need to make the procedure-call environment, binding the formal parameters of Procedure11; it is E5 in this figure:

We evaluate the body of Procedure11 in E5. It starts straightforwardly by creating a binding from name to 'Pat' in E4. Now, it evaluates self.giveRaise((age - 18) * 0.03) in E5. It starts by evaluating self.giveRaise. self is E4, so we look for a binding of giveRaise. It's not bound in E4, so we look in the parent E3; it's not bound in E3 so we look in E2 and find that it's bound to Procedure10. We are taking advantage of the fact that raises are not handled specially for this individual or for the subclass of 6.01 professors, and use the definition in the general class of 6.01 staff. The interpreter evaluates the argument to Procedure10, (60 - 18) * 0.03, getting 1.26.

It's time, now, to call Procedure10. We have to remember that the first argument will be the object through which the method was accessed: E4. So, we make a binding environment for the parameters of Procedure10, called E6:
Now the fun really starts! We evaluate `self.salary = self.salary + self.salary * percentage` in E6. We start by evaluating the right hand side: `self` is E4, `self.salary` is 100000, and `percentage` is 1.26, so the right-hand side is 226000. Now, we assign `self.salary`, which means we make a binding in E4 for `salary`, to 226000. It is important to see that, within a method call, all access to attributes of the object, class, or superclass goes through `self`. It is not possible to ‘see’ the definition of the building attribute directly from the `giveRaise` method: if `giveRaise` needed to depend on the building attribute, it would need to access it via the object, with `self.building`. This guarantees that we always get the definition of any attribute or method that is appropriate for that particular object, which may have overridden its definition in the class.

When the procedure calls are all done, the environments are finally like this:
3.7 Implementing an interpreter

From the preceding sections, you should have an informal understanding of what happens when a Python program is evaluated. Now, we want to understand it formally enough to implement an interpreter.

3.7.1 Spy

We will study a simplified language, which we call Spy, because it is a mixture of Scheme and Python. From Scheme, we take the syntax of the language, and from Python, we take the basic object-oriented programming system. Spy has a small subset of the features of these languages, but it is powerful enough to implement any computer program that can be implemented in any other language (though it could be mighty tedious).

To learn more: Interestingly, very minimal versions of several different models of computation (imperative, functional/recursive, rewrite systems) have been shown to be formally equivalent. Some have theorized them to be complete in the sense that there are no computations that cannot be expressed this way. For more information, see: http://plato.stanford.edu/entries/church-turing/

The syntax of Spy, like Scheme, is fully-parenthesized prefix syntax. Prefix means that the name of the function or operation comes before its arguments, and fully-parenthesized means that there are parentheses around every sub-expression. So, to write \(1 + \text{num} \times 4\), we would write \((+ 1 (* \text{num} 4))\).

The reason for adopting this syntax is that it is very easy to manipulate with a computer program (although some humans find it hard to read). In the following, we will assume that someone has already written a tokenizer, which is a program that can consume a stream of characters and break it into “words” for us. In particular, we’ll assume that the tokenizer can break an input Spy program down into a list of lists (of lists...of elements that are strings or integers). So the expression \((+ 1 (* \text{num} 4))\) would be converted by the tokenizer into a representation such as: 

\(('+', 1, ('*', 'num', 4))\).

Spy has the following features:

- **Integer constants:**
  
  0, 1, -1, 2, -2, ...

- **Basic built-in functions:**
  
  +, -, *, /, =, with meanings you would expect; note that in this language = is a test for equality on two integers, which returns a Boolean.
• Assignment:

(set a 7) will set the value of variable a to be 7

• Function application:

A list of expressions, the first of which is not a special word in Spy, is treated as function application, or function call. So, (+ 3 a) would return 3 plus the value of variable a, and (f) is a call of a function f with no arguments. Note that the first element can, itself, be an expression, so

(((myFunctionMaker 3) (+ 4 5))

is also a valid expression, as long as the value of (myFunctionMaker 3) is a function that will consume a single argument (which, in this case, will be the value of the function application (+ 4 5)).

• Function definition:

New functions can be defined, much as in Python or scheme. So,

(def myFun (x y) (* (+ x y) y))

defines a new function named myFun of two arguments, which can be applied with an expression like (myFun 4 9).

• If:

(if a b c) will evaluate and return the value of expression b if the value of expression a is equal to True, and otherwise will evaluate and return the value of expression c.

• Compound expression:

In Spy, the body of a function definition, and the branches of an if expression, are expected to be a single expression; in order to evaluate multiple expressions in one of those locations, we need to group them together with begin, like this:

(begin (set a 7) (set b 8) (+ a b)) .

(The reason we need begin to be a special word is so that we can distinguish this list of expressions from a function application, which is also, syntactically, a list of expressions.) The value of a compound expression is the value of its last component. So, in our example, the value of the whole expression would be 15.

Note that the names begin, set, def, and if have special meaning in Spy and cannot be used as the names of user-defined functions.

Spy also has some object-oriented features, but we will introduce those in section ??.
Exercise 3.17. Write a Spy procedure that takes two positive integers as input and returns True if the first is greater than the second. Use only the primitive functions = and + and recursion.

Exercise 3.18. Write a Spy procedure that takes \( n \) as input and returns the \( n \)th number in the Fibonacci series.

3.7.2 Evaluating Spy expressions

In this section, we will describe, in complete detail, how it is that a program in Spy can be executed. The basic operations are evaluating expressions, and assigning values to names. In the following sections, we will develop a recursive definition that allows us to compute the value resulting from any Spy program. We'll start with simple expressions, and work up to complex ones.

The spyEval function will consume an expression and some additional information, and return the value of the expression. It will have the structure of a long set of if-elif-else clauses, each of which tests to see whether the expression has a particular form and performs the appropriate evaluation.

To learn more: An interpreter is a program that takes in the text of a computer program and executes it directly. We are going to build an interpreter for Spy in this section, and when we use Python, we use an interpreter. A compiler is a program that takes in the text of a computer program and turns it into low-level instructions for a particular computer, which can later be executed. A gross characterization of the difference is that it is easier to debug when working with an interpreter, but that your program runs faster when compiled. For more information start with Wikipedia articles on Compiler and Interpreter_(computing).

3.7.2.1 Numbers

What is the value of the program ‘7’? It’s 7, of course. So, we can begin defining a function that will consume a Spy expression (described as a list of lists of strings) and return a value.
def spyEval(form, env):
    if isinstance(form, int):
        return form

    elif form[0] == 'begin':
        val = None
        for entry in form[1:]:
            val = spyEval(entry, env)
        return val

3.7.2.2 Compound expressions

As we discussed above, a list of expressions, where the first one is the word 'begin' is a compound expression, whose value is the value of the last component. But, for reasons we will see in section ??, it is important to evaluate all of the expressions, not just the last one. So, we can extend our definition of spyEval by adding another clause (the ellipsis below is meant to include the text from the previous spyEval code):

3.7.2.3 Symbols

We’ll use the term symbol to refer to any syntactic item that isn’t a number or parenthesis.

What is the value of the program ‘a’? All by itself, it is an error, because ‘a’ is undefined. So, let’s consider a more complicated program:

```
(begin (set a 6)
  a)
```

Why would we ever want to evaluate expressions and throw their values away? In a pure functional language, the answer is that we wouldn’t. But in Spy, we have assignment expressions.
They don’t even have a value, but they make a change to the environment in which they are evaluated. Before we go any further, we’ll have to take a slight detour, in order to understand the idea of environments.

**Environments**

An *environment* consists of two parts:

- a dictionary
- a parent environment (can be `None`)

The dictionary is a data structure that lets us associate values (numbers, functions, objects, etc.) with symbols; a Python dictionary works perfectly for this purpose.

We will say that a symbol is *bound* in an environment if it exists as a key in the environment’s dictionary. We’ll sometimes call the relationship between a key and a value in an environment’s dictionary a *binding*.

There are three operations we can do on an environment: look up a symbol, add a binding, and extend an environment. We’ll define the first two here, and return to environment extension later.

The *lookup* operation takes a symbol and an environment, and performs a slightly generalized version of a dictionary lookup, looking in a parent environment if the symbol is not in the dictionary:

- If the symbol is a key in the environment’s dictionary, then it returns the associated value;
- Otherwise, if this environment has a parent, then it returns the result of looking the symbol up in the parent environment;
- Otherwise, it generates an error.

The *add binding* operation takes a symbol, an environment, and a value; it adds the value to the dictionary associated with the environment, using the symbol as the key. If there was already a value associated with that symbol in the dictionary, the old value is overwritten by the new value.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>6</td>
</tr>
<tr>
<td>'b'</td>
<td>7</td>
</tr>
<tr>
<td>'c'</td>
<td>12</td>
</tr>
</tbody>
</table>

**Figure 3.1** A simple environment.
Exercise 3.19. If $e$ is the environment in Figure 3.1, what are the results of:

- $e$.lookup('a')
- $e$.lookup('c')
- $e$.lookup('d')
- $e$.add('f', 1)
- $e$.add('c', 2)
- $e$.add('a', 2)

Where do environments come from? There is one environment that is made by the interpreter, called global. It contains bindings for the primitive procedures, and is the environment in which our main program is evaluated.

**Assigning values**

So, let’s go back to the expression

```plaintext
(set a 6)
```

We’ll refer to the second and third elements of this expression (the ‘a’ and the 6) as the left-hand-side (lhs) and right-hand-side (rhs) respectively. For now, the lhs will always be a symbol. Now, to evaluate this expression in an environment, we evaluate the expression that is the rhs to get a value, and then bind the symbol that is the lhs to that value in the environment.

So, here’s our augmented definition of eval. Note that it we have added the env argument, which is the environment in which we’re evaluating the expression:

```
Spy Eval
```

```python
elif form[0] == 'set':
    (tag, lhs, expr) = form
    env.add(lhs, spyEval(expr, env))
```

So, in our simple example, lhs is the symbol ‘a’; expr is 6, which does not need further evaluation (the tokenizer has already turned the string ‘6’ into an internal Python integer 6. More generally, though, the expr can be an arbitrarily complicated expression which might take a great deal of work to evaluate.

After we have evaluated that expression, our global environment will look like this:

---

20 That usage comes because a typical assignment statement has the form $\text{lhs} = \text{rhs}$. 
Evaluating symbols

Now, finally, we can evaluate the expression \('a'\). All we do to evaluate a symbol is look it up in the environment. So, the return value of this whole program

\[(\text{begin } (\text{set } a \ 6))\]
\[
\text{a)}
\]

is the value 6.

So, now, we can add another clause to \text{spyEval}:

\[
\begin{align*}
\text{Spy Eval} & \quad \text{...} \\
& \quad \text{elif } \text{isinstance(form, str)}: \\
& \text{\quad return env.lookup(form)}
\end{align*}
\]

3.7.2.4 Functions

In the Spy language, any list of expressions, the first element of which is not a special symbol is treated as a function call.\(^{21}\) Each of the expressions in the list is evaluated, yielding a function and some number of values.

Before we go any further, we have to talk about functions in some detail. There are two types of functions: \text{primitive functions} and \text{user defined functions}. Primitive functions come built into the interpreter and are, ultimately, implemented as part of the interpreter. Spy provides several primitive functions that are bound to names in the global environment. So, to call a primitive function, all we have to do is pass the values we obtained by evaluating the rest of the expressions in the function call into the primitive function.

Function definitions

Before we can talk about calling user-defined functions, we have to talk about how to defined them. Here is an example definition:

\(^{21}\)So far, our special symbols are \text{begin} and \text{set}, and we’ll add \text{def} and \text{if}. 

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>6</td>
</tr>
<tr>
<td>'+'</td>
<td>&lt;func&gt;</td>
</tr>
<tr>
<td>;</td>
<td></td>
</tr>
</tbody>
</table>
(def fizz (x y)
  (+ x y))

It is a list of four elements:

1. `def`, which is a special symbol, indicating that this expression is not a function call, and therefore requires special handling;
2. the name of the function being defined (‘fizz’ in this case);
3. a list of symbols, called the formal parameters (in this example, (‘x’, ’y’); and
4. a function body, which is a Spy expression, in this case, ’(+ x y)’

Not very much happens at function definition time. We construct a data structure (typically an instance of a Function class) that simply stores three components:

- the formal parameters
- the function body
- the environment in which the function definition was evaluated

Then, we add a binding in the environment in which the function was defined from the function name to this function structure. With this understanding of function definition, we can add another clause to `spyEval`:

```
Spy Eval
...
elif form[0] == 'def':
  (tag, name, params, body) = form
  env.add(name, Function(params, body, env))
```

Here is a picture of the global environment after the definition of `fizz` is made. Note the binding from the name ‘fizz’ to a Function instance which, itself, contains a reference to the environment.
Function calls

Now, we can see what happens when we actually call a function. Let’s assume that the function \texttt{fizz} from the previous section has been defined, and now we want to evaluate

\[(\texttt{fizz 6 8})\]

in the environment \texttt{env}. This is the trickiest part of the whole interpreter. First, we evaluate each of these expressions, getting: a \texttt{Function} instance (which we created when we defined the \texttt{fizz} function), the number 6 and the number 8.

Now, we make a new environment. Let’s call it (imaginatively) \texttt{newEnv}. The dictionary part of \texttt{newEnv} is used to associate the symbols that are the formal parameters of the function (in the case of the \texttt{fizz} function, ’\texttt{x}’ and ’\texttt{y}’) with the actual values being passed into the function (in this case, the numbers 6 and 8). The parent environment of \texttt{newEnv} is the environment that we stored with the function when it was defined (not the environment in which the function is being called!!). In this case, \texttt{fizz} was defined in the global environment. And so, the parent pointer is also the global environment. (Later, we may see some examples where this pointer is different, in a way that matters.)

Here is a diagram of the entire environment, at this point:

```
now eval

Fizz
\texttt{(x, y)}
(+ x y)
\texttt{y 8}
\texttt{x 6}
```

Now that we have this new environment, we evaluate the expression that is the body of the function, in this case, ’\texttt{(+ x y)}’ in \texttt{newEnv}. This will all work out roughly as you expect it to: during the process of evaluating the body, we will have to evaluate the symbols ’\texttt{+}’, ’\texttt{x}’ and ’\texttt{y}’. We will find ’\texttt{x}’ and ’\texttt{y}’ in the dictionary of \texttt{newEnv}; and we will find ’\texttt{+}’ in its parent environment, which is \texttt{global}. So, we’ll have the function object that is bound to ’\texttt{+}’, and the number 6 and the number 8; then, we’ll call the primitive function on these values, and end up with the result 14.

Here is our augmented definition of \texttt{spyEval}:
As required, this process evaluates the first expression in the list to get a function. Then it evaluates the body of that function in a new environment that maps the function’s formal parameters to the results of evaluating all the rest of the expressions in the list, and whose parent is the environment stored in the function.

A good way to understand what is going on with spyEval is to show a trace of its behavior. So, when we evaluate the following expression:

\[
\begin{align*}
\text{(begin} & \text{ (def } \text{fizz} \ (a \ b) \\
& \quad (+ \ a \ b)) \\
& \quad (\text{fizz} \ 3 \ 4))
\end{align*}
\]

we can print out the arguments that are passed into spyEval (though we have left out the environment, to keep things from being too cluttered), and the result of each call. Each time it is called recursively, we print out the arguments and results with one level more of indentation:

```
args: ['begin', ['def', 'fizz', ['a', 'b'], ['+', 'a', 'b']], ['fizz', 3, 4]]
args: ['def', 'fizz', ['a', 'b'], ['+', 'a', 'b']]
result: None
args: ['fizz', 3, 4]
  args: fizz
  result: <__main__.Function instance at 0x7b1e18>
  args: 3
  result: 3
  args: 4
  result: 4
  args: ['+', 'a', 'b']
    args: +
    result: Primitive <built-in function add>
    args: a
    result: 3
    args: b
    result: 4
    result: 7
  result: 7
result: 7
```

Here is another version, showing the basic part of the environment, but not the parent environment.
3.7.2.5 If

The last special type of expression in Spy is a conditional, of the form:

```
(if (= x 3)
  (fizz x 10)
  (+ x 4))
```

It might seem, at first, that it would be okay to implement if as a primitive function, similar to +. But there is an important reason not to do so. Consider the definition of the factorial function, in Spy:

```
(def factorial (x)
  (if (= x 1)
    1
    (* x (factorial (- x 1)))))
```

The most straightforward application of this function, (factorial 1) will get us into trouble. Why? We would start by evaluating 'factorial' and '1', and getting a function and 1. So far so good. Now we make a new environment, and evaluate the body in that environment. That requires evaluating all of the elements of the body in that environment. So, we'd find that 'if' evaluates to a primitive function, that (= x 1) evaluates to True, and that 1 evaluates to 1. Then, we'd need to evaluate that last expression, which is itself a function call. That, in itself, is no problem. But we can see that eventually, we'll have to evaluate factorial in an environment where its argument is -1. And that will require evaluating factorial of -2, which will require evaluating factorial of -3, and so on.
The most important thing about 'if' is that it evaluates only one of its branches, depending on whether its condition is True or False. Without this property, we cannot stop recursion.

So, now we can see what to do with an 'if' expression. We'll call the second, third, and fourth elements the condition, then and else parts. We start by evaluating the condition part. If it is True, we evaluate the then part, otherwise we evaluate the else part.

It is pretty straightforward to add this to our eval function; but note that we had to put it before the function application clause, so that we can keep if expressions from being evaluated as if they were function applications.

Here is the whole interpreter.

```python
def spyEval(form, env=globalEnv):
    if isinstance(form, int):
        return form
    elif isinstance(form, str):
        return env.lookup(form)
    elif form[0] == 'begin':
        val = None
        for entry in form[1:]:
            val = spyEval(entry, env)
        return val
    elif form[0] == 'set':
        (tag, lhs, expr) = form
        env.add(lhs, spyEval(expr, env))
    elif form[0] == 'def':
        (tag, name, params, body) = form
        env.add(name, Function(params, body, env))
    elif form[0] == 'if':
        (tag, condition, ifBody, elseBody) = form
        if spyEval(condition, env)
            return spyEval(ifBody, env)
        else:
            return spyEval(elseBody, env)
    elif isinstance(form, list):
        f = spyEval(form[0], env)
        return spyEval(f.body,
            Environment(f.formal,
                [spyEval(x, env) for x in form[1:]],
                f.environment))
    else:
        Error("Illegal expression: "+str(form))
```

Yay! Now we have a complete Spy interpreter. Well, we still need implementations of the Environment and Function classes, but that's not very much more work. We can implement those classes as exercises.
3.7.3  Object-Oriented Spy

We can add a simple object-oriented facility to Spy, which is modeled on Python’s OOP facility, but is somewhat simpler. The crucial idea here is that classes and instances are both environments, of exactly the same kind that we have been using to support binding and function calls in basic Spy. The dictionary part of the environment supports the binding of attribute names to values within the instance or class; and the parent environment part allows an instance to be connected to its class, or a class to its superclass, so that attributes that are not defined within the instance may be found in the class or superclass.

We only have to add two new syntactic features to the language: attribute lookup and class definition.

3.7.3.1  Attribute lookup

In Python, when you want to get the value of an attribute a from an instance obj, you say obj.a. In OOSpy, we’ll say (attr obj a). Remembering that an instance is an environment, all we have to do to evaluate such an expression is to look up the symbol ‘a’ in the environment ‘obj’.

Referring to the second part of this form as the object part and the third as the name part, we can add a clause for attr expressions to our interpreter. The name will always be a single symbol; but the object part can be a general expression (we might, for example, call a function bigFun that returns an object, yielding an expression like (attr (bigFun 3) x) to get the attribute named x from the object returned by bigFun). This means that we have to evaluate the object part, using our standard evaluation function. Now, we can add a clause for attr expressions to our spyEval:

```
Spy Eval
...
elif form[0] == 'attr':
    (tag, objectExpr, name) = form
    return spyEval(objectExpr, env).lookup(name)
...
```

3.7.3.2  Class definition

All the rest of our work happens at class definition time. Here is a simple OOSpy class definition:

```
(class SimpleClass None
    (begin (def init (self v))
        (set (attr self v) v))
    (def getV (self)
        (attr self v))))
```

For reference, it is roughly equivalent to the Python:
class SimpleClass:
    def init(self, v):
        self.v = v
    def getV(self):
        return self.v

It has four parts: the special symbol `class`, the class name SimpleClass, the name of a superclass (in this case, there is no superclass, so we write None), and a class body, which is a Spy expression. Our simple Spy implementation doesn’t have an equivalent to Python’s `__init__` facility, in which the specially-named procedure is called automatically. So we call the method `init` without underscores to make that clear.

Here is a picture of the environment that we would like to have result from this definition:

![Environment Diagram]

Note that Spy differs from Python here in an important way: in Python, the environment stored in a method is always the module environment; in Spy, to avoid having a special case, it is the class environment in which it is defined.

Also, rather than binding SimpleClass directly to the environment for the class, we will bind it to a procedure that makes a new environment whose parent is the class environment. This procedure can be called directly to create a new instance of the class.

There are three steps involved in processing this class definition.

1. Make an environment for the class  In this step, we just make a new empty environment, and set its parent to be the class specified as the superclass of this class. If the superclass is None, then we use global as the parent environment, so that global variables and primitive function definitions will be accessible from the class body.

2. Evaluate the class body in the class environment  In this step, we use our regular Spy evaluation mechanism, to evaluate the class body, in the new environment. In our SimpleClass body, we define two methods; these definitions will appear in the class environment but will not, for example, be accessible directly from the global environment.

3. Make a constructor function  Finally, we define a new function, with the same name as the class, in the environment in which the class definition is being evaluated (not the class environment; if we did that, nobody would ever be able to find the constructor!). The constructor
function has the job of making a new instance of this class. What is an instance? An environment. And a brand new instance of any class is simply an empty environment, whose parent environment is the class environment.

After defining SimpleClass, we can use it as follows:

```
(begin
  (set a (SimpleClass))
  ((attr a init) a 37)
  ((attr a getV) a))
```

This is roughly equivalent to the Python

```python
a = SimpleClass()
a.init(37)
a.getV()
```

This may look a little bit strange. Let’s go step by step. First, we make an instance by calling the constructor function. That’s pretty straightforward. Now, the value of variable `a` is an instance of the SimpleClass class, which means that it is an environment.

Now, what about this expression?

```
((attr a init) a 37)
```

Let’s try to work through it the way the interpreter would. It is a list, and the first element is not a special symbol, so it must be a function call. That means we should evaluate each of the forms in the list.

First we evaluate `(attr a init)`, which is a list starting with the special symbol `attr`, so we know it’s looking up an attribute in a class. It evaluates the expression `a`, getting the SimpleClass instance, then looks for the `init` attribute, which yields the `init` function that we defined inside the class body.

Evaluating the second and third elements of `((attr a init) a 37)` yield our SimpleClass instance and the number 37.

Now, we’re ready to do a function call. Let’s do it carefully. The formal parameters of the function we’re calling are (`'self'`, `'v'`). So, we make a new environment, in which those parameters are bound to the SimpleClass instance and to 37, respectively, and whose parent environment is the environment in which `((attr a init) a 37)` is being evaluated. Now, we evaluate the body of that function, which is

```
(set (attr self v) v)
```

in this new environment. Let’s call this environment `E1`. In order to understand what’s going on here, we have to go slowly and carefully. Here’s `E1`: 
First, we recognize this as a set expression. But, so far, our set expressions have always had a single symbol as their second element. Now, we will have to generalize it, so the second element of a set expression can also be an attr expression. Let’s consider this kind of expression, in general:

\[(\text{set} \ (\text{attr} \ x \ y) \ z)\]

In this form, \(x\) can be any expression that, when evaluated, yields an instance; \(y\) must be a single symbol; and \(z\) can be any expression that, when evaluated, yields a value. The result of this form is to set attribute \(y\) in the instance resulting from evaluating expression \(x\) to have the value resulting from evaluating the expression \(z\).

So, we need to evaluate \((\text{set} \ (\text{attr} \ self \ v) \ v)\) in the environment \(E1\). We start by evaluating \(self\), and getting the \texttt{SimpleClass} instance; we look for an attribute named \(v\) there and, not finding one, make a new one. Now, we evaluate the third part of the set expression, which is \(v\), in the \(E1\), and get the value 37. Finally, we set attribute \(v\) of our instance to 37. Yay. Here’s the new state of the environment:
One thing to notice is that Spy, as we have designed it, doesn’t have Python’s special facility that automatically uses an object as the first argument of a method, when that method is accessed via that object. So, we have to pass the object in again, explicitly, as the first argument.

Evaluating the expression `((attr a init) 37)` works similarly (you should be sure you understand how), and returns the value 37.

So, after all that, we can add the last clause to our OOSpy interpreter, for handling class definitions. Note that we also have to extend the code for handling `set`, so it can deal with setting attributes of instances.

```python
... elif form[0] == 'set':
    (tag, lhs, expr) = form
    (targetEnv, name) = lhsEval(lhs, env)
    targetEnv.add(name, spyEval(expr, env))

eelif form[0] == 'class':
    (tag, name, super, body) = form
    if super == 'None':
        super = globalEnv
    classEnv = Environment(parent = super)
    env.add(name, Primitive(lambda : Environment(parent = classEnv)))
    spyEval(body, classEnv)
...

def lhsEval(lhs, env):
    if isinstance(lhs, list):
        (tag, objectExpr, name) = lhs
        return (spyEval(objectExpr, env), name)
    else:
        return (env, lhs)
```

**Primitive** is a class that represents a primitive function in Spy. Its initializer just takes a procedure as an argument: in this case, it is a procedure that makes a new environment whose parent is the the class environment.

We’re done with a whole object-oriented language!
Exercise 3.21. Write an OOSpy class definition for an Averager class. It should have a method `addValue` that allows you to “add” a numeric value to the set of data it has seen and method `getAverage` that returns the average of all of the values that have been added so far.