18

Motivation

Yesterday, we introduced the environment model of computation as an alternative to the earlier substitution model of computation. This newer model was introduced to overcome some of the shortcomings in the substitution model that arose primarily because we began to consider state and stateful objects as significant players in our programs. Barring implementation details, the environment model of computation is a fairly accurate description of how Python and many other languages establish relationships between variables and the values that they refer to. In fact, for project 4, you will get a chance to implement the model yourself, but as employed in Scheme, a dialect of Lisp, and the language of this course until a year ago.

The environment model is also meant to provide a more accurate answer to some of the other questions that you may have been asking yourselves throughout the semester, and that you may have built a particular kind of intuition towards. For instance, we saw this piece of code in lecture 4 of week 1:

```python
def funny_function(a, b, c):
    def a_funny(b):
        return (a * b) + (b / a) + (a ** b)
    return a_funny(b) / a_funny(c)
```

We used this code to motivate inner and local helper functions. Back then, we told you that the variable b for the inner function a_funny shadowed the variable b for the outer function funny_function. This meant that whatever the argument provided as b to funny_function, it would be ignored in favor of whatever was provided as b to a_funny. However, whatever the argument provided as c to funny_function, it
would still be considered \texttt{c} in the internal function \texttt{a\_funny}. At this point, we had some notion of “scopes”: we knew when a variable was “alive”, when it was not, and when it would be shadowed by another variable of the same name.

Similarly, for an even more basic piece of code:

```python
x = 5
def square(x):
    return x * x
```

we built an intuition that the variable \texttt{x} in the inside of \texttt{square} is different, somehow, from the variable \texttt{x} on the outside, the one set to \texttt{5}. This intuition explained by the \texttt{square} function did not always return 25.

The environment model is an attempt at formalizing this kind of intuition, to ensure that our intuition does not lead us astray in more complicated cases, and to ensure that we are thinking about our code in a similar manner as the interpreter is dealing with our code. With this more formal model in hand, we can consider tackling more interesting problems. We can also begin to implement an interpreter ourselves, which understands and works according to this formal model, the subject of our fourth module on interpretation, which begins next week. Why can we do this? We can because the environment model is simply a collection of exact rules and an algorithm that applies those rules to understand expressions and statements. Most, if not all, of our programs so far have been mere implementations of algorithms.

Bear in mind though that the environment model is nothing more than a bookkeeping device: it allows us to keep track of what variable names map to what values, and in what environments variables are “alive” and/or shadowing other variables.

## 2 Rules

We have provided the rules as the last page of these lecture notes. Feel free to tear them out and read them as we move along, and to also use them for other contexts, such as exams, discussions, and homework assignments.

## 3 Higher Order Functions that Return Functions

Yesterday, we saw an example of nested functions. We also saw one example that dealt with higher order functions that returned functions:

```python
>>> def make_adder(x):
...     def add_x(y):
...         return x + y
...     return add_x
```

CS61A Summer 2012: Tom Magrino and Jon Kotker, with
Joy Jeng, Eric Kim, Stephen Martinis, Allen Nguyen, Steven Tang, Albert Wu
First, we define `make_adder`. Remember what a definition of a function entails: we
draw a "double bubble" with the first bubble pointing to both the arguments and the body of
the function, while the second bubble points to whatever frame was current when the
function was made. Since the global frame was current when the function was made, the
right bubble points back to the global frame. In a sense, you can think of the right bubble
as reminding us what frames were current at the time of its creation, so that – as needed –
we can look up various variables that were in place (and may have since changed) when
the function was created, and thus what the function assumes will be available whenever
it is finally called.

That’s it though: we only draw the bubble. We don’t evaluate the body of the function,
since we haven’t called it yet. At this point, our diagram should look like:

![Diagram](image)

Now, we finally call `make_adder` when determining what value should be assigned to
`add_5`. How do we do that? Well, there was one thing that the substitution model got
right, which was that the operator is first evaluated, to understand what the operator is
(Is it a function? is it a variable? If the latter, then Python would complain. But first, it
needs to understand what the operator is.). Here, according to our environment diagram,
`make_adder` is a function, which is good, since we are going to call it and apply it on to
its operand(s). Then, we evaluate each of its operands from left to right: 5 is primitive
and evaluates to the number 5 in any frame.

We now know what our operator and operands are, so let us call the operator on the
operands. What happens when we call a function? We make a new frame, which extends
whatever frame the right bubble of the function extended. Here, the new frame E1 will
extend the global frame G. Now, we bind the arguments of the function to the values
provided to the function: here, the argument x is bound to the value 5. Finally, we make
E1 the current frame.
The order here is important: we first bind the arguments to the values, and then we make the frame the current frame. This order serves to remind us that, while we are still binding arguments to values, we have not yet evaluated the body, and so the frame is not yet current – sometimes, the process of figuring out what values are associated with what arguments can be so involved that we can forget that the suite of the function still has to be evaluated! So remember, bind arguments first and then make the new frame the current frame.

We like to keep track of current frames by maintaining a stack of frames on the side. The name stack implies the stack data structure, where elements are added and removed from the same end of the data structure: here, we are adding new frames to the side (or bottom) of the stack, and removing the frames as we are done with function bodies. The global frame is always to the left (or top) of the stack.

Phew. You know this model is formal when there is already so much to talk about, and we have only reached the beginning of the body suite of the make_adder function! (I kid.) The body tells us to define a new function called add_x. We do the needful: we bind the name add_x to a double bubble, the right bubble of which now points to the current frame, which is frame E1. Notice what would have happened if we had pointed to the global frame instead: we would have totally ignored the variable x, which is integral to how make_adder works! Looking ahead a bit, we need some way to “remember” x whenever we finally use the function add_x, because x was “active” and within scope when the function add_x was birthed, and that is a variable that add_x expects will be available whenever it is eventually called. The variable x is also integral here since it serves to generalize the generation of adders, which was one of our motivations for higher order functions in the first place.
Again, we do not go into the body of `add_x`, since we have not yet called it. However, we do return whatever the name `add_x` points to, back to whatever needed it, which was `add_5`. We remove E1 from the stack of frames, since we have already finished working through the body of `make_adder`. The variable `add_5` in the global frame now points to a function. We now have the following diagram:

Notice that we now have two names pointing to the same function `double bubble`. This is not as weird as you might initially think. It is exactly what the code is doing: two names are now bound to the same function value!

So now, we evaluate `add_5(7)`. As before, we evaluate the operator, here `add_5`. We follow the binding in the current frame (the global frame) to find that `add_5` is a function! Cool. Okay, so now we evaluate the operands: well, 7 evaluates to 7 in any frame. Now, we call the operator on the operands. Again, we have our new frame E2 extend the frame that the second bubble points to. Once we have finished evaluating, we end up with this final diagram:
4 Announcements

- HW9 is due **Friday, July 20**.
- Project 3 is due **Thursday, July 26**.
- Midterm 2 is next **Wednesday, July 25**.
  - One page of your own notes.
  - We will give you both the old notes sheet we provided on last midterm and an additional one for new material.
  - Group section will be at the beginning this time, and will again take 15 minutes.
  - We will test material from the beginning through the end of this week.
- Midterm 1 solutions are posted.
  - Regrades are due by July 26th.
  - Attach an explanation to the front of your exam explaining what you think should be regraded and why.
  - We reserve the right to regrade the entire exam.

5 Higher Order Functions that take Functions as Arguments

When we talked about higher order functions, we also found that we could take other functions as arguments. Let us try that with our new model. Consider the following example:

```python
>>> def call_on_5(fn):
...     return fn(5)

>>> def square(x):
...     return x * x

>>> call_on_5(square)
25
```

First, we evaluate the definitions, after which we have the following environment diagram:
Remember, we have not yet called any of the functions.

We are now going to evaluate the expression on the last line. We first evaluate the operand, which, in the current frame (the global frame) happens to be a function! The operator also evaluates to a function. Now, we perform the call to `call_on_5`. We bind the arguments: the variable `fn` is bound to the function that was named by `square`. Again, we have two bindings for the same function. Now, we are done binding the arguments, and so we make the new frame `E1` the current frame, and we have the following diagram:

Aside: This should hopefully help you understand why functions can be passed around as data. All we are merely doing is binding variables to whatever was being passed into functions: does it matter if that particular value happened to be a function? A function, just like any other piece of data, is still a sequence of 0s and 1s inside the computer: it’s still a thing, but a thing that has the ability to take in other pieces of data – even other functions – and work on these pieces of data. It’s similar to how you would talk about any other machine, really: you point to it and you give it a name. So what if it’s not running: the machine still exists, and you are referring to it. This is one of the powerful ideas proposed by John Von Neumann in his stored-program model, which revolutionized computer science: earlier, computers were configured for very specific functions. With the new stored-program model, the data and the instructions for manipulating the data were stored in RAM, and all the computer had to do was read these instructions when processing the data.
Back to the environment diagram: we are now in the body of `call_on_5`. We are asked to evaluate `fn(5)`. We consider the operator and operand separately. What does the operator evaluate to? It evaluates to a machine, as we would hope – otherwise, Python will tell us that we are trying to call something that is not a function. The operand evaluates to 5. So now we call the function `fn` on 5. We create a new frame, which extends ... the global frame, since that is where the right bubble of the `square` function points! (This is again where the function was “birthed”.) We’re not extending the current frame! We bind its argument `x` to 5 and make E2 the current frame.

![Environment Diagram](image)

We evaluate the body to obtain 25 and return that to whomever was asking, while removing E2 from the current frame stack. We are now done with the body of `call_on_5`, which means that we return 25, while removing E1 from the current frame stack. We are back in the global frame, and the prompt prints the result 25. Ta-dah!

6 **Recursion**

Before we consider ourselves done with functions, let us examine an example with recursion, where we have more than one call to the same function in the diagram. Let us use the following code:

```python
>>> def fact(n):
...     if n == 0:
...         return 1
...     return n * fact(n - 1)
```

CS61A Summer 2012: Tom Magrino and Jon Kotker, with
Joy Jeng, Eric Kim, Stephen Martinis, Allen Nguyen, Steven Tang, Albert Wu
>>> fact(3)
6

By this point, we know what happens up until the recursive call. Right before the recursive call, we have the following diagram:

So how does a recursive call work? Really, we treat it as if we were calling some other function and make the frames for the recursive calls the same way as we did before. We look up the function, make a new frame that extends the frame pointed to by the second bubble, and evaluate. After we are done, we’ll have:

7 Lists, Tuples, and Dictionaries

Before we tackle a new piece of Python syntax, we will take a brief moment to explore how we would represent Python lists, tuples, and dictionaries (and not their immutable counterparts). Consider the following interaction with the Python interpreter:

```python
>>> my_list = [1, 2, 3, 4]
>>> my_tuple = (1, 2, 3, 4)
>>> my_dict = {'I': 1, 'V': 5, 'X': 10}
```

The environment diagram would look like this:
Lists and tuples, as before, are represented with boxes and pointers. To distinguish between the two, we write a small $L$ to the side of the list and a small $T$ to the side of the tuple to distinguish between the two. A dictionary looks a lot like a frame, since there are bindings between keys and values, and so we represent dictionaries with ovals to distinguish between the two.

Our current model should now allow us to explain why we can mutate lists and dictionaries in functions without having to return them. Let us draw the environment diagram for the following piece of code, which continues from the code above:

```python
>>> def make_first_five(l):
...     l[0] = 5
>>> make_first_five(my_list)
>>> my_list
[5, 2, 3, 4]
```

When we evaluate the call to the function `make_first_five`, as always, we create a new frame. In this frame, we bind `l` to whatever `my_list` evaluated to, which is a list. Now, we make the new frame the current frame.

Inside the body of the function, we are asked to modify the first element of whatever `l` is to 5. We notice that `l` points to the original `my_list` and proceed to update that list directly. This is why when we are finally done with the function call, we notice that our original list has changed.
The same rule applies to dictionaries. Of course, since we cannot modify tuples, we must construct new tuples every time we concatenate tuples. For instance, consider the following piece of code:

```python
>>> my_tuple = my_tuple + (2, my_list, 4)
```

In evaluating an assignment, we always evaluate the right side first. We notice that we need to determine what `my_tuple` evaluates to: it is a tuple. When we “add” two tuples, we make a new tuple that is as big as the total size of the two elements, and stick the new elements into this new tuple. The environment diagram would now look like:
Notice that the new tuple still “contains” the list my_list: it points to the list, since that is what my_list evaluated to.

Aside. In the context of Python’s magic methods, when we “add” two tuples together, we are actually calling the __add__ method on the first tuple object. This method has been written such that it reads the incoming tuple, evaluates the elements of this tuple, and then sticks these elements into the new tuple returned by the __add__ method.

8 Nonlocal State

In the syntax of Python that we have seen so far, we find that the statement
<variable> = <value>

has two meanings:

x = 5
x = x + 1

In the first instance of the statement, we are defining a new variable called x with its value set to 5. In the second instance, we are updating the value of the variable x with a new value of 6.

According to the rules that we have presented so far, if the variable being assigned to is not already in the current frame, we consider the statement <variable> = <value> to be a definition of a new variable, and not an update of a pre-existing variable. This is true even if the variable exists in another frame that can be reached by the current frame.

For example, consider the function count_calls, which takes in a function of one argument. It returns what the function would usually return, but also prints out the number of times that the function was called. For example, here is the intended interaction with the function:

>>> counted_square = count_calls(square)
>>> counted_square(2)
Function called 1 times.
4
>>> counted_square(3)
Function called 2 times.
9
>>> counted_square(4)
Function called 3 times.
16

Here is a possible solution:
>>> def count_calls(fn):
...     num_times = 0
...     def counted_fn(arg):
...         num_times = num_times + 1
...         print("Function called " + str(num_times) + " times.")
...         return fn(arg)
...     return counted_fn

>>> counted_square = count_calls(square)

What does the environment diagram look like after the definition of the `count_calls` and the evaluation of `count_calls` on the `square` function?

Notice that we have a new frame $E_1$ that extends the global frame, which contains a binding to the variable `num_times`. This variable stores the result of the last call to the function `fn` provided. This is the variable that we would like to update inside `counted_fn`.

Now, we evaluate the expression `counted_square(2)`. We create a new frame $E_2$ that extends frame $E_1$, since the right bubble of the function pointed to by `counted_square` points to frame $E_1$. In this frame, we bind the argument `arg` to the value provided, make $E_2$ the current frame, and then start evaluating the body of `new_fn`. Just before we begin to evaluate the body of `new_fn`, however, Python signals an error:

UnboundLocalError: local variable 'num_times' referenced before assignment
Wait, why? Well, `lookup` is different from assignment. When we look a value up, we
know that we follow frames and the pointers to their parents all the way to the global
frame, at which point – if we cannot find the variable – Python signals a `NameError`.
However, for an assignment statement, if we cannot find the variable in the current frame,
we assume the user meant a fresh assignment instead of an update, and so we make a
fresh assignment in the current frame.

This is exactly what Python does, but it goes the extra mile to make things more efficient
for us under the hood. (This is not generally true of other languages.) What it does, at
the beginning of the function, is that it goes through the code of the function and sees what
variables we will be needing and updating for the duration of the function. It notices that
we will need the value of `num_times` to update the value of `num_times`. However, to
Python, we would not have specified the value of `num_times` yet by that point! Why?
Because it looks up the value of `num_times` in the current frame, and it can’t find any!
More technically, the value of `num_times` is `unbound`. So, Python, to save us some de-
bugging effort, decides to warn us that we’ll be assigning to a variable that we haven’t
defined yet. This is clearly not what we had intended.

To ensure that Python also looks at assignments in parent frames, we introduce the key-
word `nonlocal`:

```python
>>> def count_calls(fn):
...     num_times = 0
...     def counted_fn(arg):
...         nonlocal num_times
...         num_times = num_times + 1
...         print("Function called "+str(num_times)+" times.")
...         return fn(arg)
...     return counted_fn
>>> counted_square = count_calls(square)
```

The statement that specifies which variables are `nonlocal` should be done before those
variables are used. This new piece of syntax allows our code to behave as expected.

We also add a new rule to our environment model, which notes that if a variable is defined
to be `nonlocal`, we first make sure that it has not already been defined in a parent frame,
before we decide to make a fresh new assignment to that variable.
Caveat: nonlocal does not work if the variable that you are trying to update is in the global frame. In this case, we would use the keyword global instead.

```python
>>> x = 5
>>> def foo():
...     global x
...     x = 6
>>> foo()
>>> x
6
```

9 Conclusion

Today, we saw more extensions to the environment model to include notation for lists, dictionaries, and tuples, and also to include the possibility of updating a variable outside the current scope. We now have all the information and rules that we need to construct environment diagrams for most of the Python functions and code that we have seen so far, with the exception of classes and objects. Tomorrow, we will see that we do not really need new notation for classes and objects, or new rules, since we are able to construct the basics of the OOP system in Python using just functions and dictionaries.
Environment Model Rules

- **Looking Up Variables**
  1. Start with the current frame.
  2. Look in the frame for a binding from the variable name to a value.
  3. If it is not in the current frame, repeat the search in the parent frame.
  4. Once a value is found, return to the current frame.

- **Variable Assignment**
  1. Evaluate the right hand side, and continue drawing the environment diagram as you evaluate.
  2. Create a binding in the current frame from the variable name to the value of the right-hand side, only if there is no binding for the variable in the current frame. Otherwise, update the binding in the current frame. *Special case:* If the variable is nonlocal, update the binding in a parent frame, if there is a binding in a parent frame that is not the global frame. Otherwise, make a binding in the current frame.

- **Function Creation**
  1. Create a double bubble.
  2. Point the first bubble to a list of parameters and the body of the function.
  3. Point the second bubble to the current frame.
  4. If this was a `def` statement, create a binding from the function name to the double bubble in the current frame.

- **Calling Functions**
  1. Find the value of the function.
     - If this was a built-in function, treat it like “magic” (where the built-in function is defined in global and not shown).
     - Otherwise, locate the double bubble for the user defined function.
  2. Find the value of the arguments.
  3. - If this was a built-in function, treat it like a black box and find the resulting value returned.
     - Otherwise, if it is a user defined function:
       (a) Draw a new frame.
(b) Create bindings in the new frame from the argument names to the values found earlier.

(c) Have the new frame extend the frame that the second bubble points to.

(d) Make this new frame the current frame (this is called “stepping into” the new frame) and evaluate the function body pointed to by the first bubble.

(e) Once the function body is evaluated, return with the value and restore the current frame to what it was before we stepped into the new frame.