CS61A Lecture 2

Friday, August 30th, 2019

Announcements

- Homeworks are not graded based on correctness, but on effort. If you prove you spent good faith effort on the homework, you will get the marks.
- Labs are designed to be less challenging, but they are not easy by any means.
- Lectures are always available online; this semester it will also be available on edX with checkpoint questions regarding your understanding.
  - This edX option is available for the first time this semester.
  - Make sure you watch Lectures 1, 2 and 3 before doing Lab 1 next week.
- If you think you have enough experience to skip 61A, fill out the info form regarding CS47A on the website, which only requires you to do the exam and a few projects from the last few weeks of class.

Names, Assignment, and User-Defined Functions

These features are available in almost every programming language, but of course, we will do it in Python. The terminal is where you can do things in a command line.

You can run an interactive `python3` interpreter by typing `python3` in your terminal, which will look like this:

```
Python 3.7.1
>>>
```

The `>>>` indicates you can start typing your command there.

Certain commands, like `pi`, must be imported as they are not always available.

You can clear your terminal with `Ctrl/Cmd+L`. You can assign any name to any value you want, except for a select range of reserved names.

Assign a name using an equal sign, name on the left, and value on the right. One thing that surprises people is that a name is assigned to a value, not the expression. For example,

```
>>> from math import pi
>>> radius = 10
>>> area = pi * radius ** 2
>>> area
314.15926
>>> radius = 20
```

This does not change the value of area to the area of a circle with radius 20. It doesn't remember that area is equal to pi times the radius squared. It only remembers the value is 314.15926.

You need to reassign the value by running `area = pi * radius ** 2` again.

You can even assign names to replace built-in function names like `max` and `abs`. For example,

```
>>> max(2,3)
3
>>> max = 7
>>> max
7
```

The easiest way to “fix” this is to quit Python and start again from scratch. Python sometimes lets you reach deep into the interpreter and recover the built-in functions, but you do not need to know this in the course.

Other than changing built-in functions, Python will also let you define your own, new functions. For example, the function `square`:

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

Remember earlier when we said that you have to re-run `area` when `radius` is updated? We can fix that with a function:

```
>>> radius=10
>>> def area():
...     return pi * radius ** 2
>>> area()
314.15926
>>> radius = 20
>>> area()
1256.6371
```

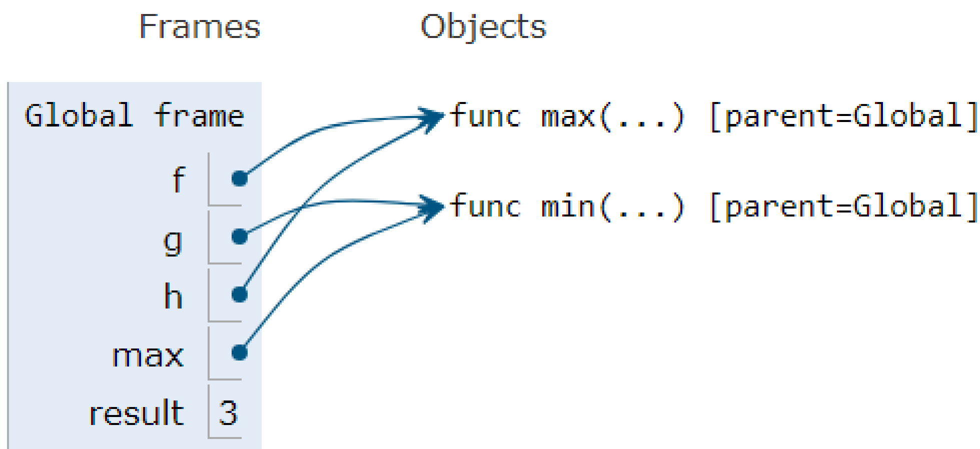
So far we have seen two kinds of expressions, primitive expressions and call expressions. Primitive expressions are those like numbers, strings and so on, while call expressions call on a function.

Environment Diagrams

> **Discussion**
> What is the result of this sequence?
>
> ```
> >>> f = min
> >>> f = max
> >>> g, h = min, max
> >>> max = g
> >>> max(f(2,g(h(1,5),3)),4)
> ```

It’s confusing right? Surely there’s an easier way to track these names? As humans, we have been draw boxes and arrows to keep track of things. In Python, environment diagrams are used. An environment is what Python uses to keep track of which variables are usable in which functions.

Environment diagrams visualize the interpreter’s process.



On the left is the code, and the right is frames.

Statements and expressions, and the arrows indicate evaluation order. Each name is bound to a value, and each name can only appear once. If a name is assigned to one value, then to another, the old value is deleted and replaced with the new one.

You can access an automatically drawn environment diagram with tutor.cs61a.org.

Assignment Statements

They bind the name on the left to the value of the expression on the right. Rules:

1. You evaluate the expression to the right of the equal sign, from left to right, ignoring the left first.
2. Bind all the names to the left of the equal sign to those resulting values, **in the current frame**.

Returning to the Discussion Question, we can use an environment diagram to see that `max` is now bound to the `min` function, so we are trying to find what is the smallest value found in the group, which is 3.

> **Why isn't the assignment `f=min` instead `f=min()` ?**
> Doing so would bind the name `f` to the result of computing the minimum of nothing (an error!), which would run an error. We want to bind f to the name min, which is a function.

Defining Functions

This is a big deal, and it is the key to abstraction. Assignment statements are a means of abstraction, giving a name to something without having to learn the lower-level details.

Functions that you define yourselves give name to a series of expressions. Function definition is a powerful means of expression.

A def statement consists of the words def, the function signature, which indicates how many arguments a function takes, then the body defines the computation performed when the function is applied.

Execution procedure for def statements:

1. Create a function with the signature `<name>(<formal parameters>)` .
2. Set the body of that function to be everything indented after the first line.

3. Bind `<name>` to that function in the current frame.

Notice that the execution procedure doesn't have you calculate the expressions within the function, and that the calculation occurs in the execution procedure of the call function.

Calling user-defined functions

Here is the execution procedure for call expressions:

1. Add a local frame, forming a new environment.
2. Bind the function's formal parameters to its arguments in that frame.
3. Execute the body of the function **in the new environment**.

Names are bound to values, not complete expressions.

Notice:

```
>>> def square(x):
...     return mul(x,x)
>>> square(3-5)
4
```

The value of `x` in your environment diagram would be -2, not 3-5.

Every expression is evaluated in the context of an environment. An environment is a sequence of frames. A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

So far, the current environment is either:

- the global frame alone,
- or a local frame, followed by the global frame.

Basically, when evaluating a name in a local context (e.g. within a function), look for it in the local frame, and if it's not there, look in the global frame. For example:

```
>>> x = 2
>>> def print_x():
...     x = 3
...     print(x)
>>> print_x()
3
>>> def print_x_2():
...     y = 1
...     print(x)
>>> print_x_2()
2
```

This function `print_x_2` looks for `x` in its local frame, doesn't find it, and looks in the global environment, where `x` is 2! It cannot see names local to the `print_x` frame.