CS 61A CSM
Fall 2017

The Structure and Interpretation of Computer Programs
Kartik Kapur

Midterm 1 Review

1 Introduction

This worksheet will serve as a brief review of everything that has occurred in CS61A over the last few weeks. It is not comprehensive, but it should provide the high level overview that is required to gain intuition for most problems.

2 Functions

Functions, in the most basic sense, are a block of code, that can be used to perform some type of action. Functions have multiple uses, and can be used as return values, nested definitions, and arguments.

2.1 Structure

```
1 def hello(hi):
2       ...
3       return bye
```

In the above function, "hello" is the function name, hi is the argument and bye is the return value. Now that we've gone over the basics of functions, let's describe all of their use cases.

2.2 Functions as arguments

Functions can be used as arguments. Let's take the following example:

```
1 def eatFood(makeFood):
2       makeFood()
3       print("I have eaten!")
```

Here we have a function *eatFood*. In this, we have an argument, *makeFood*. We call the function *makeFood* and then say that "I have eaten!". Depending on the food that we want to eat, we will have a different method to make it. However, regardless of how we make the food, we will eat it the same way- with our mouths! Instead of making a new function for eating every type of food, we have a function for eating. This function takes in a function that makes our food, which makes a more universal function.

2.3 Returning Functions

Say we wanted to make a function that allows us to eat cook some food given 1 ingredient and a set amount of time. At runtime, our function would take in a "mystery ingredient" and we would compare all the foods

```
1 func1 = makeFood(ravioli, x, 2)
2 func2 = makeFood(ice cream, x, 2)
3 func3 = makeFood(ravioli, x, 2211)
```

You can see why this can get a bit tiring, so instead of doing this, we can return a function. For the above example, we can make a function that takes in 3 arguments, the 2 food items, and the amount of time you want to cook them for. This looks as follows:

```
1 def checkFood(food1, time):
2       return makeFood(food1, mystery, time)
```

Now our functions can be rewritten as

```
1 func1 = checkFood(ravioli, 2)
2 func2 = checkFood(ice cream, 2)
3 func3 = checkFood(ravioli, 2211)
```

The change isn't too obvious in this scenario; however, in times when you want some similar construction with different inputs (the x value), this process can save a lot of time. An example will come up when we deal with Nested Functions.

2.4 Nested Functions

Now that we have dealt with functions as return values and arguments, we will now deal with functions within functions. This concept can seem a bit overwhelming, but I promise, it will get a bit easier. Nested functions allow us to not have all the methods public to the global frame and it allows us to use the output of a specific function as the only possible input for a function.

```
1 def makeRavioli(time):
2     def putSauce(flavor):
3         print("put sauce")
4     print("boiled it")
5     return putSauce
```

Here we have a nested definition of *putSauce*. We would not want people to put sauce on brownies, so it makes sense that this function would be within the *makeRavioli* function. Additionally, if someone had some other pasta, they may want to have a *putSauce* method that acts differently. Having nested definitions allows us to have methods with the same name and purpose that have different functionality.

3 Recursion

You are in a theater and want to know how many rows are in front of you but you can only see 1 row in front of you. To solve this problem, you ask the person in front of you how many people are in front of you?. They ask the person in front of them and this continues until reaching the person in the first row, they respond 0 to the person in the 2nd row. The person in the 2nd row knows there are 0 rows in the row in front of him, so he adds 1 to 0, 1, and tells guy behind him. That guy adds 1 to 1, 2, and tells guy behind him. This process repeats until you get back a number. This process is a real life application of the concept of *recursion*.
Recursion, in the most basic sense, is a function that calls itself, or other functions, and each time it does this, it

simplifies the input problem until, it it simple enough to be solve trivially. Recursive algorithms consist of 2 main parts, the base case(s) and the the recursive call. A *base case* is a conditional statement that that terminates the algorithm when invoked. The *recursive step* is the call to the function that has a reduced input in order to solve the problem.

```
1 def countdown(num):
2     if(num ==1):
3         return("bye")
4     return countdown(num-1)
```

In the above function, line 2 contains the base case, where, if the num variable equals 1, then the algorithm terminates. Otherwise, we call the the function again on a smaller input.

3.1 Recursion between 2 functions

We have dealt with functions that call themselves; however, in recursion, sometimes we call other functions, and use them to make our input smaller.

```
def countdown(num):
    if(num ==1):
        return("bye")
    return countdown(num-1)
def divide(num):
    if(num == 1):
        return("hi")
    return countdown(num/2)
```

In the above example, the recursive step of *countdown* returns the output of the function *divide* with an input of num - 1. The recursive step of *divide* returns the output of the function *countdown* with an input of num/2. These functions can be particularly useful when we want to alternate between 2 steps.

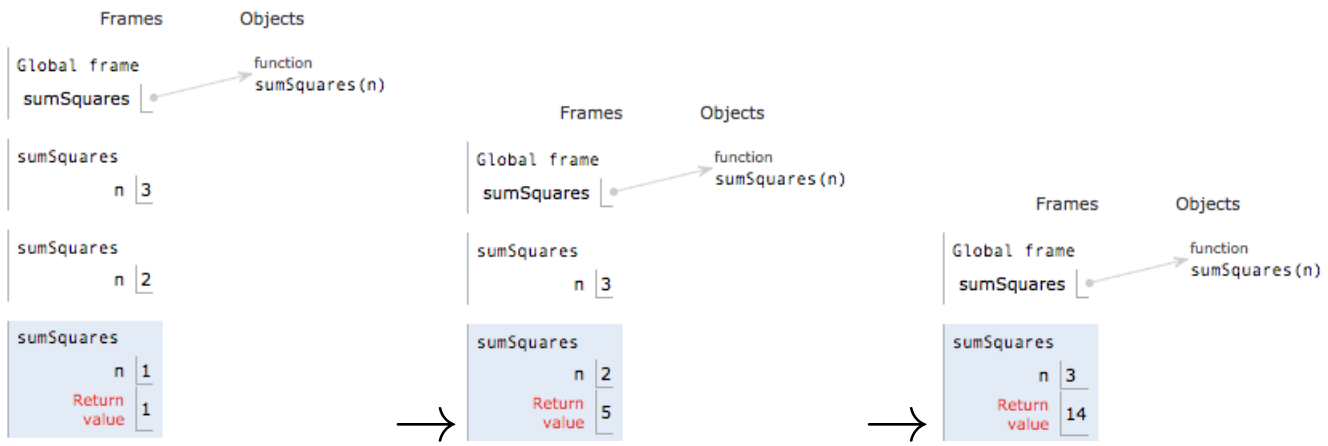
3.2 Recursion going up

Recursion is not always as simple as just going down, sometimes, once your algorithm terminates, we need to "go up" as well. This can occur when there are steps after or including the recursive call.

```
def sumSquares(n):
    if(n == 1):
        return 1
    return pow(n,2) + sumSquares(n-1)
```

In the above function, we are attempting to find the sum of the squares of the first n natural numbers. At each step on the way down, we add n square to the return value of the sumSquares(n-1). For example, for take sumSquares(3). The return value of this function is 9 + sumSquares(2), for sumSquares(2) the return value would be 4 + sumSquares(1), and for 1, the base case holds so the return value is 1.
From here, we essentially perform a series of substitutions. We substitute 1 for sumSquares(1) which gets up 4+ 1 = 5

for sumSquares 2. We then substitute this value into sumSquares(2) and get 9 + 5 =14 for sumSquares(3) return value. At this point, we have exhausted all the steps that we must perform and we return 14.
This process is equivalent to the following environment diagrams



Note that the first frame is for the return value of *sumSquares(1)*, the 2nd for *sumSquares(2)*, and the 3rd for *sumSquares(3)*.

3.3 Tree Recursion

Tree Recursion is the process when a function that calls itself more than once during one step. For example:

```
def walkSteps(n):
```

```
2       if(n <= 2):
3           return n
4       return walkSteps(n-1) + walkSteps(n-2)
```

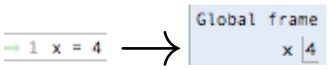
In this function, we are counting the amount of ways that we can walk on a certain path of distance n steps, where you can walk either 1 step or 2 steps at a time. We will walk through the example of n =3. We return walkSteps(2) + walkSteps(1). We then return n for each one of these case, since from 2 there are 2 ways to reach the end by walking 2 steps once or 1 step twice, and from 1 there is 1 way to reach the end. We add these 2 cases and get a total of 3 ways to walk the path.
At each step, you see the paths that will occur if you take 1 step or 2 steps. Once you reach a distance of 2 or less steps from the end, you simply return n.

4 Environments

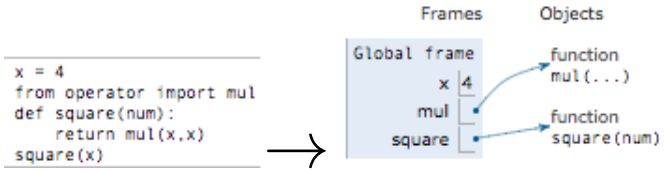
In CS61A we deal with *environment diagrams* which help us visualize how we can walk through a program, we will discuss how to think about each step in an environment diagram.

4.1 Basics of Environments

Variables in Python can be thought of as a storage location for anything, from a function to a number to a string. Variables take the following form



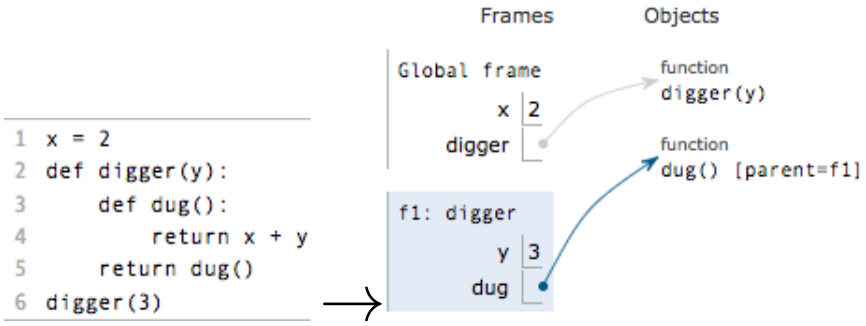
The item on the left side of the equals is the variable and the item on the right side is the value that it is taking on.
There are 2 types of major types functions in Python, *user defined functions* and *Non-user defined functions*. A user defined function is a function that the user builds that takes in some set variables.



In the above set of calls, *mul* is something defined from operator and *square* is a user defined function. Note how in the frames, mul takes in an arbitrary amount of arguments labeled as "..." whereas *square* takes in only 1 argument.

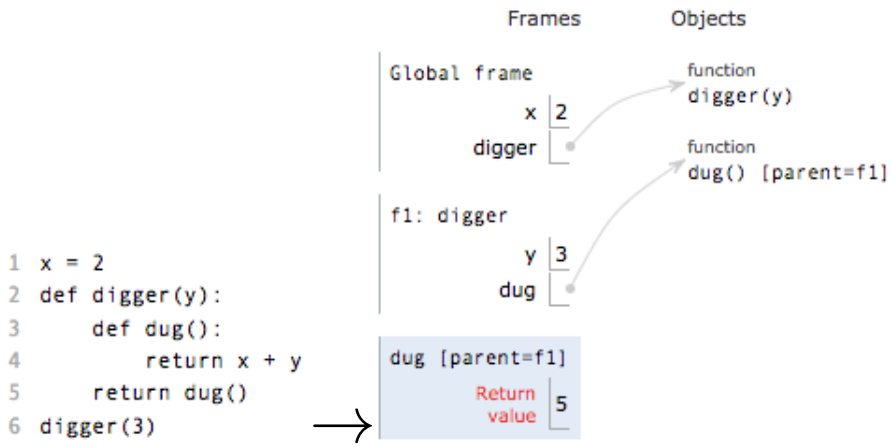
4.2 Frames

Frames can be thought of as a way to keep track of a function call in progress. We will examine a function and pick apart frames and their attributes.



The *Global Frame* is a frame that has no parent, and every variable or function is either in the global frame or some

sub frame. A *Parent Frame* is the frame which a function is defined in. In the above function, x and *digger* are defined in the Global Frame. The function *dug* has a parent in frame 1, or our call to *digger* with the argument 3.



When we completed our call in *dug*, we see that even though we did not define x and y in the frame, it still managed to complete without error! We will discuss why this is in the next lesson.

4.3 Scope

In the earlier section, confusion may have arisen when somehow our variables gained access to other ones? How do we know which variables we can use? Well there are a fixed set of rules about variables and their accessibility.

- You can only access variables in scopes that are above the current frame.
- You cannot modify the variable in a lower scope. You can only get its value
- A variable can only be defined once in a scope (no variables with 2 names defined in the same scope).
- If 2 variables have the same name and are defined in different scopes, the one that is closer will be used as the value
- A function's variables have the value of the scope that it was defined in.

That was a lot of rules, so lets go over some examples when the rules would hold.

```
1 def eat():
2     hunger = false
3 eat()
4 appetite = hunger
```

This block of code would error out because appetite is in the frame above the frame that hunger would be defined in. As a result, we cannot set appetite equal to it. Mainly because there is the parent frame has no clue about what is below it. A way to get around this is as follows.

```
1 def eat():
2     hunger = false
3     return hunger
4 appetite  = eat()
```

We modified our code so that appetite would be equal to the return value of *eat*. Since eat returns false, appetite would be false.

```
energy = 10
def work():
    energy -= 
work()
```

This above code would error out because we cannot directly modify the energy variable in a frame below it. What you can do is make a variable, set it equal to energy and then modify that. That is done as follows.

```
energy = 10
def work():
    lifeforce = energy
    lifeforce -= 
work()
```

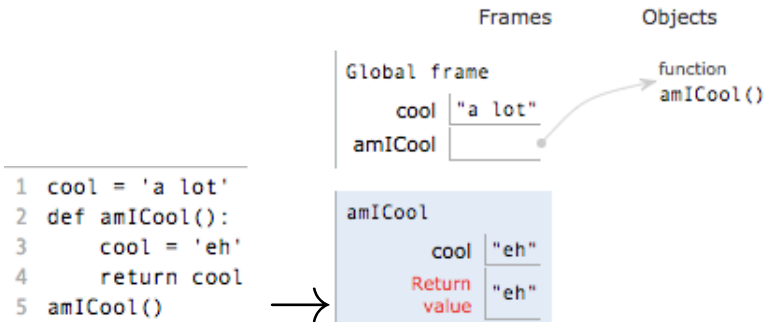
This would run properly; however, if you wanted to change the value of energy, you would need to do something similar to out fix in the prior problem.

```
energy = 10
def work(x):
    x -= 
    return x
energy = work(energy)
```

Here we can modify the energy variable by setting it equal to the result of the work method.

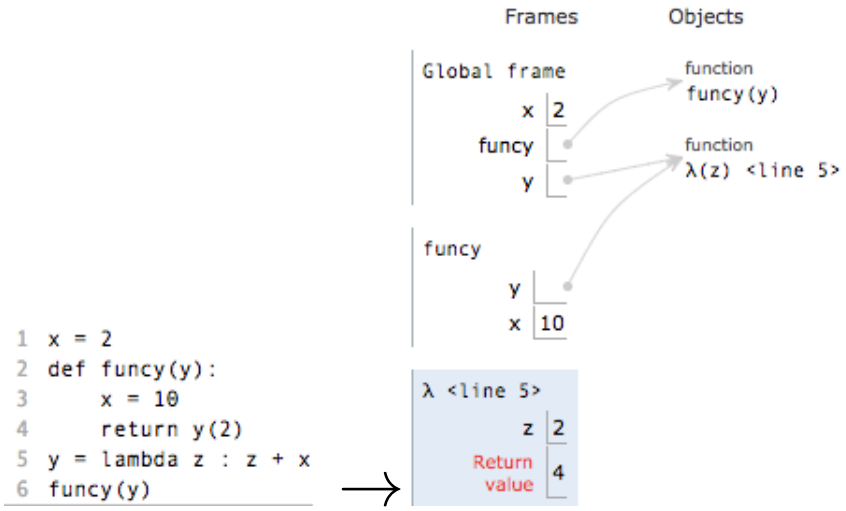
```
x = 2
x = 3
```

In this case, no real error occurs. Basically all that is happening is that the variable x is associated with 3 and not x once the program is completed.



In this example, we have the variable *cool* defined in the global frame. Then when we call *amICool* it returns ”eh.

That is because we defined a second cool in that frame that is closer to our call of cool.
The last rule is the one that most people consider a bit more confusing.



In the above function, we have a lambda function that is defined in the global frame. We then pass it into out function

funcy and attempt to evaluate it. We pass in the value 2 as the argument for z; however, when it comes to choosing the value for x, **we do not go to the frame that the function was evaluated in, instead we pick the value of x based off where the function was defined**. If x was not defined in the global frame, there would be an error.

5 Misc.

Here I will discuss miscellaneous categories that are in the first unit of CS61A that tend to be a bit tricky

5.1 Lambda Functions

Lambda function are a concise way to write definition statements.

```
def func1(x):
    return 10
```

The above example is equivalent to

```
lambda x : 10
```

Realize that there is a 1 to 1 correspondence between lambda function and def statements. Rewriting lambda's in a much more well known form can really aid in the understanding of these types of problems.

5.2 The Print Function

The print function is one that is really unique. It displays something to the user output; however, the value of a print is always None. For example:

```
def print10():
    x = print(10)
    return x
```

This function would first print 10 then bound x to the return value of the print statement. This is always None, regardless of the input to the print function.
Additionally, returning and printing a string would yield different outputs. Printing a string would always result in the

said string having no quotes around it; however, if you returned a string, there would be quotes around it. For example:

```
def returnStr():
    return "hello"
```

returns 'hello'. While

```
def printStr():
    print "hello"
```

prints hello. You do not really need to know why this occurs, but it is an interesting fact to keep in mind.

6 Butter Nutter

(a) Label the parts of the following function:

```
1  peanut = butter = "crunchy"
2  jelly = "strawberry"
3
4
5  def sandwich(butter, bread):
6      if (len(bread) > len(butter) and len(butter) != 0):
7          return sandwich(butter[1:], bread + butter[0])
8      return bread
9
10
11 sandwich(peanut + " ", jelly)
```

Solution:

```
1  peanut = butter = "crunchy" #global variables
2  jelly = "strawberry" #global variable
3  def sandwich(butter, bread): #function name
4      if (len(bread) > len(butter) and len(butter) != 0): #conditional
5          return sandwich(butter[1:], bread + butter[0]) #recursive call
6      return bread #return statement
7
8
9  sandwich(peanut + " ", jelly) #function call
```

(b) What does the above code snippit return?

Solution: strawberrycrunchy

7 Goats and Lambs

(a) Lambdas are very niche functions. They are used to write short, one line def statements. Write the following lambda statement as a normal def statement function. lambda x: lambda y : lambda z: x + y + 1

Solution:

```
1  def func1(x):
2      def func2(y):
3          def func3(z):
4              return x + y + 1
5          return func3
6      return func(2)
```

(b) Write the following function as a lambda:

```
1  def fib(n):
2      if(n < =1):
3          return n
4      else:
5          return fib(n-1)+ fib(n-2)
```

Solution: fib = lambda n: n if n <=1 else fib(n-1)+fib(n-2)

(c) Within 4 lines create a function or set of functions that do the following: You are given an integer input. If n, the number of digits, in the integer is odd find n factorial. Otherwise, find the nth fibonacci number

Solution:

```
1 dig = lambda x: 1 if x < 10 else 1 + dig(x//10)
2 fib = lambda x: x if x <=2 else fib(x-1) + fib(x-2)
3 fact = lambda x : 1 if x<=1 else x*factorial(x-1)
4 func1 = lambda x: fib(dig(x)) if dig(x)%2 == 0 else fact(dig(x))
```