Introduction to Computer Science

By: Kartik Kapur and Kunal Kapur

Contents

1 Introduction **0**
1.1 Acknowledgments . . . 0
1.2 Why Computer Science? . . . 0
1.3 Python . . . 0

2 Getting Started **1**
2.1 The Fundamentals . . . 1
2.2 Building Blocks of Computer Science . . . 2
2.2.1 Primitives . . . 2
2.2.2 Functions . . . 2
2.3 Revisting the Assignment Statement . . . 3
2.4 Checkpoint 1 . . . 5
2.5 Checkpoint 1 Solutions . . . 6
2.6 Nested Expressions . . . 7
2.7 Frames and Environment Diagrams . . . 7
2.7.1 Scope of Variables . . . 10

Chapter 1

Introduction

1.1 Acknowledgments

This book is primarily based off of University of California Berkeley's Computer Science 61A and borrows heavily from **Composing Programs**. However, the curriculum created by this will be geared towards students in high school and middle school so that they can build the foundation required to succeed in Computer Science.

1.2 Why Computer Science?

Computer Science is now a very prominent field in the world. With increased automation in the world, computer science is becoming more and more important. The concepts that one learns in computer science can be applied to a series of logic puzzles and can heavily aid in critical thinking abilities.

1.3 Python

The programming language that we will be going over in this book is Python. Python is an English-esque language and a lot of the logic that holds in English will also tend to hold in Python.

To run Python, if you work on a Mac, simply go to the terminal and type in "python3". However, we may not always want to work on the terminal, since it be a lot more difficult to compose larger programs. As a result, we can work with an IDE or text editor, my personal favorites are **PyCharm** and **Atom**. IDE's let you do some really useful processes such as debugging your code and running in a simplistic eye appealing environment; however, text editors will allow you to have a light weight interface where you can simply code and run on the terminal.

Whenver there are examples, run python3 in your terminal and copy the input. It will help with your understanding of the material. If there are some parts that are vague, go to **Pythontutor**. It is an amazing website that helps you visualize your code.

Chapter 2

Getting Started

2.1 The Fundamentals

In Computer Science in general, programs are nothing more than instructions. These instructions may initially seem very overwhelming, almost a foreign language; however, all of these instructions fit into one of the following 2 categories.

1. Computing a value
2. Carrying out an action

Expressions are generally the vocabulary used to discuss a computation. When we say that Python evaluates an expression, we mean that it computers the value of that expression. For example,

```
1 >>> 3+5
2 8
```

We typed 3+5 into the Python interpreter. It then evaluated the expression and outputed the value, 8.

On the other hand, a **Statement** describes carrying out a specific action. An example of a Statement is the **Assignment Statement**. The Assignment Statement associates some variable with the value of an expression. This is done in the following manner, _____ = _____. When this occurs, the item on the left will get the value of the expression on the right. For example:

```
1  >>> a = 3 + 4 + 5
2  >>> a
3 12
```

We can see that we associated a to the value of "3+4+5". Since we evaluated the expression on the right, a was associated with 12. Once we called a we got that value as an output.

Another statement that we will go over is the **Import Statement**. Oftentimes, there are **packages** that contain information that we can use. Instead of recreating the wheel each time we want to make a program, we can import information from these packages. Import statements take the following form: from _____ import _____.

A time where this could be useful is when we want a mathematical value, for example pi. Let's do try the following

```
1  >>> pi
2 NameError: name 'pi' is not defined
```

We obviously did not want an error to occur, we wanted the value of pi. So what we can do is utilize the handy math package and do the following.

```
>>> from math import pi
>>> pi
3.141592653589793
```

Now let's try a brief example putting together everything that we went over in this lesson.

```
>>> from math import sqrt
>>> a = sqrt(16)
>>> a
4.0
```

In the above example, we imported the square root function from the math package. Following this, we set a equal to the square root of 16. Since we set a equal to the value of that expression, when we call a we get the output of 4.0.

We will go more in depth with statements and expression in the following lessons, but for now make sure that you have an understanding of the basic definitions.

2.2 Building Blocks of Computer Science

This section will go over the basic elements of Computer Science. We generally want to work with some forms of **data**. Our data can be just about anything: numbers, words, types of food, you name it! Just dealing with data by itself can be a bit boring, we want to be able to modify our data. To do this, we can create **functions**.

2.2.1 Primitives

Some basic types of data that we will discuss early on are numbers like 10,20, 1.1233, or -11000. The next type is the string, which we define as a set of characters in quotations. Below are a few examples of our terminal output if we input some primitives.

```
>>>1.1
1.1
>>>'hi'
'hi
>>>"hi"
'hi'
```

It may seem confusing that "hi" and 'hi' are two different expressions; however, we can easily generalize that any values surrounded by quotes (either single quotes or double quotes) is a string. It is important to realize that you cannot mix and match these quotes- that is you cannot do "hi' or 'hi".

One last primitive that we will go over is the **boolean**. Boolean values are true or false values. An example follows:

```
>>>True
True
>>>False
False
```

Every single value in Python can be characterized as either True or False, and as we progress through the text, we will see what values are "True" values and which one are "False" values.

2.2.2 Functions

Now that we have defined some basic types of data, we can define some ways to modify them. There are 2 types of functions that we will discuss, **in-built functions** and **user-defined functions**.

Before we get into detail about the various types of functions, we should go over the syntax of a function. To call a function, we would use the following syntax: **function_name(argument_1, argument_2...argument_n)**. Just as a note, an argument is a specific input for a function.

In-built functions are functions that are already defined for us. What this means is that we don't know how the function itself works, we just know that it outputs will output something corresponding to our input.

```
1 >>>from math import sqrt
2 >>>sqrt(64)
3 8.0
```

In the above case, we imported the square root function. After that, we did that, we will call the function on the argument 64 and then get the output of 8. We do not know exactly what goes on in this function, but we do know that it works and that it outputs the proper square root of a number. This is a very basic example; however, later on, we will see that it is important that we do not know how functions are defined.

We will now go over user-defined functions. To define a function, we use the keyword **def**. We can do this as follows: **def function_name(parameter_1, parameter_2)**. A parameter is a variable in a method definition. When we call a function, we pass in arguments, these arguments take the place of the parameter. Arguments are basically specific values while parameters are placeholder variables. We will define a basic function.

```
1 >>>def add_2(x)
2             return x+2
3 >>>add_2(10)
4 12
```

The first thing that you may notice is that in the line after our def statement, we have a group of spaces/indents. The indented block is what we call the **body of the function**. We define the start and end of the function based off the indentation level. Anything nested by either 1 tab or 4 spaces , your choice, is considered to be in the body of a function. It is very important that you do not mix up tabs and spaces- there are very lively debates about which is better all over the internet, it is up to you to choose a side.

The next syntax that we will go over from the above function is the **return statement**. Return statements terminate your function and associate them with some value. In this case, we want to associate the value of 2+x with the function add_2. In line 3, we perform a function call to add_2 with an argument of 10. The argument 10 takes the place of the parameter x inside the function add_2 we would then return 10+2 which would return 12. An important clarification to make is that you can only return values within functions. We will go over why later in this chapter.

2.3 Revisting the Assignment Statement

When we set a variable equal to some value, we say we are **binding** that variable to a value. However, we can change the values of variables by setting the variable equal to another value, this process is called **reassigning** a variable. Earlier, we could assign some primitive values to variables, and that's great, but oftentimes we need to use the results of some data manipulation in order to design what we will assign to a variable. So let's take a step into that direction. Just as a refresher, here is the assignment statement in action.

```
1 >>>a = 5
2 >>>a
3 5
4 >>>b = 10+2
5 >>>b
6 12
```

But now that we have functions, our horizons have expanded! We can now use the assignment statement to assign variables to the output of functions. Let's go through an example.

```
>>>def multiply_2(x):
        return x*2
>>>a = multiply_2(10)
>>>a
20
```

In the above code, we created a function multiply_2 and with the parameter x. We then called the function multiply_2 with the argument 10. The return value of this is 20 so we would assign the variable a to it. When we call a, the value would be 20. It is extremely important to remember that when we evaluate assignment statements, the right side is always evaluated before the left side. That means that only when are all the function calls complete on the right do we assign the value to the left.

We know we can make variables equal to the output of functions, but let's try something a little more ground breaking: making variables equal to functions themselves.

```
>>>def multiply_2(x):
        return x*2
>>>a = multiply_2
>>>a
<function multiply_2 at 0x100662e18>
```

We set the variable a equal to a function; however, since we did not evaluate the function by passing in an argument, the evaluated statement on the right of the equals sign is the function itself. That means that a will be equal to the actual function multiply_2. Then why, when we ask for a do we not just get the output multiply_2? Remember this question, we will address it in the following 2 sections.

2.4 Checkpoint 1

1. Take the following snippet of code. What are the values of the variables a and b?

```
1 a = 1
2 b = 1
3 b, a =  a+b,b
```

2. What would be the value of z after this code has run?

```
1 def func1(x,y):
2       return x*y
3 z = func1(10,2)
```

3. What is the value of p after this code has run?

```
1 def func2(a):
2       return a*a
3 p = 10
4 p = func2(p)
```

4. Why will the following code not run properly?

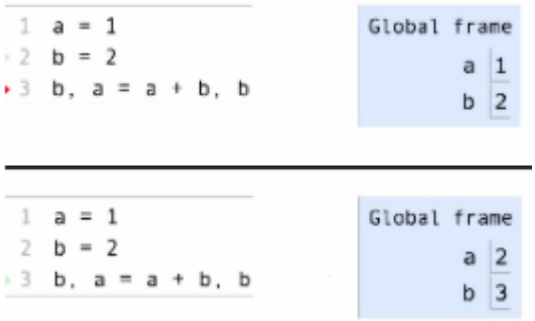
```
1 def hi(y):
2       return 2
3 z = hi(z)
```

5. What does the following code display?

```
1 def adder(x,y):
2       return x+y
3 f = adder(2)
```

2.5 Checkpoint 1 Solutions

1. **a = 2, b =3**. Reasoning below



   Initially, a is bound to the value 1 and b is bound to the value 2. Things get a bit tricky on line 3. We have the expression "b,a = a+b,b". Using our rules, we evaluate the right side of the equals sign before the left side. So we evaluate a+b, getting us the result of 3. We don't yet reassign b to equal 3 yet though. We first evaluate the expression to the right of the comma which is just b, or 2. We then set a equal to 2 and b equal to 3.

   Important takeaways from this problem is that we evaluate the right side of the equals sign before the left side. After we or on a side, we evaluate from left to right. We never reassign the variables on the left of the equals until every single expression is done being evaluated.

2. **20**. As always, we evaluate the right side of the equals sign before the left side. We run the function func1 with the arguments 10 and 2. The function return a value of 20 so we bind z to the value of 20.

3. **100**. This initially may seem a bit weird since we're making p equal to a function that uses p as an argument. The pattern that has been emerging in the past few problems continues. We evaluate the right side of the equals sign before the left, in fact don't even look at the left side of the equals side, it tends to be confusing. We see p is equal to 10 initially, we then run func2 with the argument of p, which is 10. The function returns a*a which is 10*10 or 100, so func2 evaluates to 100, which means that p is bound to the value of 100.

4. It may be a bit confusing to see why this code doesn't run, z is defined after all so it feels like the code should run. Since we evaluate the right side of the equals sign before we evaluate the left side, z is actually not defined at the moment we are passing in some nonexistent variable into a function and setting z equal to it. Basically, what this means is you cannot make a variable equal to a function if the argument has yet to be instantiated, or bound to a value.

5. This code would display an error. The reason why is because the function we are attempting to run takes in 2 arguments but we are only passing in 1. This function cannot run without 2 arguments

2.6 Nested Expressions

When we pass variables into functions, sometimes we want to be able to pass in some modified variable. This means that we need to perform more than one function on this variable. This is the motivation behind **Nested Expressions**. When we have nested expressions, we would have something of the following form: **func1(func2(variable), func3(variable))**. We will follow the simple rule of evaluating the most nested item first. Let's walk through an example.

```
1 def adder(x,y):
2       return x+y
3 def square(z):
4       return z*z
5 a = adder(square(2), square(3))
```

So we have 2 functions defined, adder and square. We then have some variable a that we know will be bound to the result of calling adder on the square of 2 and the square of 3. We will follow our basic rule of evaluating everything to the right of the equals sign before the left, which means we need to evaluate adder(square(2), square(3)).

We will begin by evaluating the most nested expressions from left to right. First we evaluate square(2) which is 4 then we evaluate square(3) which is 9. This leaves us with the function adder called on 4 and 9 as follows which can be thought of as follows:

```
1 def adder(x,y):
2       return x+y
3 def square(z):
4       return z*z
5 a = adder(4,9)
```

We would call the function on it, which would result in a getting bound the the value of 13. In general, when dealing with nested functions, it is important to simplify the problem. Begin by evaluating nested expressions and "replacing" the operands in the parent calls with the value of the nested expression. This will help clear your mind and allow you to do these problems systematically.

2.7 Frames and Environment Diagrams

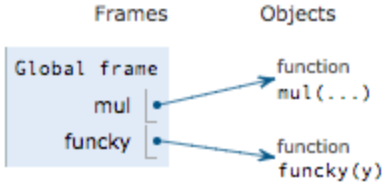
We earlier said that variables had some stored value. But to keep track of all these variables and functions, we need to store them within some structure. We will call an **environment** a place in a computer's memory where we store all this information. Environments consist of a series of frames, each frame refers to a new function call. More on this in a few.

So far, we have dealt with somewhat basic functions and calls. With many function calls and variables, it becomes imperative that we have some organized method of keeping account of them. We will be using a visualization called **Environment Diagrams** to keep track of bindings, the association of a name to some value. Frames will be represented as boxes with variables within them. This is all a bit complicated to write in words, so we will go through this with pictures. Take the **following function**. The picture is one from **Pythontutor**, the website where we will display environment diagrams. The green arrow refers to a line that has executed and the red arrow refers to what will be executed at the next step.

Python 3.6

```
1 from operator import mul
2 def funcky(y):
3     y = 10
4     x = 5
5     return mul(x,y)
6 z = funcky(1)
7 z+=1
```

Edit code | Live programming



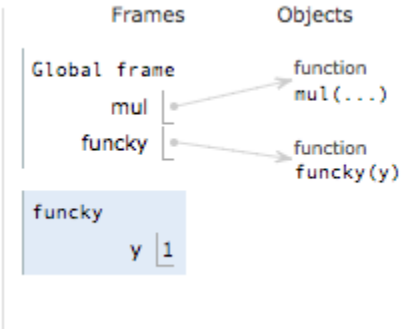
We start off in what is called the **Global Frame**. The Global Frame is not that special from other frames other than the fact that assignment and import statements can only occur in the Global Frame. We start off in the first line in our Global Frame and assign mul to the pre-define function "mul". In the next line we define a' user defined function, funcky. We assign a value of the function funcky to the name "funcky". We do not actually evaluate the function yet because we have not reached a line where we have a function call to funcky. Note how on the "Objects" side of our image, the user-define function funcky has a parameter y wheras for our in-built function, mul, we have some "..." in parentheses. The reason for this is that the function funcky can only take in 1 value whereas the mul function can take in an arbitrary amount of arguments.

Additionally, we have the idea of **Parent Frames**. Whenever a function is called, it has a parent. The parent frame is the frame in which the function is defined. In this case, funcky's parent is the Global Frame.This concept will come more into play in the next lesson.

Python 3.6

```
1 from operator import mul
2 def funcky(y):
3     y = 10
4     x = 5
5     return mul(x,y)
6 z = funcky(1)
7 z+=1
```

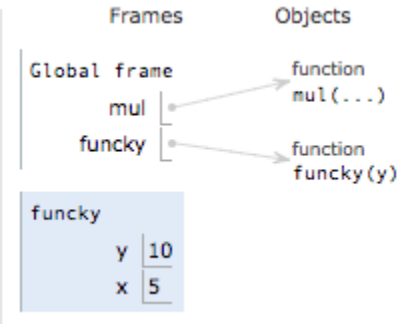
Edit code | Live programming



Python 3.6

```
1 from operator import mul
2 def funcky(y):
3     y = 10
4     x = 5
5     return mul(x,y)
6 z = funcky(1)
7 z+=1
```

Edit code | Live programming



To evaluate the call funcky(1), we will create a frame for this specific function call. The y value would start off as 1, as that is the value we initially pass in our actual function call. However, in the first step of funcky, y gets redefined to be 10. Following this, x gets defined to be 5.

We move onto line 5. There is a call to the function mul with arguments x and y. We look in the current frame and see that x is defined to be 5 and y is defined to be 10. We evaluate the function call, which results in the value of 50. The reason why we created a frame for the function funcky and not not the mul function is because mul is an in-built function wheras funcky is user defined. We can actually walk through what user defined functions do because we have knowledge on how they work; however, we have no clue how Python chooses to implement its in-built functions.

There is nothing more to do in this frame because we just returned, essentially exiting from the frame. We then return to the Global frame and set z equal to the value of the return value of funcky(1), which, as discovered earlier, is 50.

In the final line, we use a syntax that has not been seen yet. We have z+=1, the "+=" essentially means that we will add whatever is on the right side to the item on the left side. Or in terms that may be easier to understand:

```
1  z = z + 1
```

Since z was originally 50, we add 1 to it getting 51. Following this, we reassign z to be equal to 51.

We will not usually go this indepth for environment diagrams; however, we feel that it was useful to go

through it for this one example to get the basic idea of how they work. Whenever referring to a snippet of code in an environment diagram, we will provide a link in blue so you can work with the code interactively.

2.7.1 Scope of Variables

Now that we know what frames are, there comes an interesting question. Can we access all variables from anywhere in the program? The answer is no! These frames designate a sort of hierarchy and we will go over them in this section. An important concept for scopes is the idea of Parent Frames. Just as a review, a frame's parent is the frame that the function was defined in.

1. Parent frames cannot make direct references to variables in child frames. This makes intuitive sense. You cannot be sure that a function call would occur, as a result, you have no knowledge on if a frame executed or not.

2. Child frames can view variables in parent frames; however, they cannot directly edit those values

We will go into more indepth examples.