Introduction to Computer Science

By: Kunal Kapur and Kartik Kapur

Contents

Chapter 1

Python and Syntax Fundamentals

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,

```
>>> 3+5
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:

```
 >>> a = 3 + 4 + 5
 >>> a
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

```
 >>> pi
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.

```
>>>from math import sqrt
>>>sqrt(64)
 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. If a function runs its completion and has no return statement then the associated value would be None. 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 and Comparisons

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.

```
>>>a = 5
>>>a
 5
>>>b = 10+2
```

```
>>>b
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.

We know we can assign variables to the outputs of functions but what can functions output? Well we can basically make a function output anything! From strings to numbers to other functions. Anything that a variable can be assigned to can be returned by a function.

2.4 Checkpoint 1

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

```
a = 1
b = 1
b, a = a+b,b
```

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

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

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

```
def func2(a):
    return a*a
p = 10
p = func2(p)
```

4. Why will the following code not run properly?

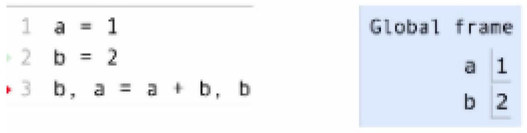
```
def hi(y):
    return 2
z = hi(z)
```

5. What does the following code display?

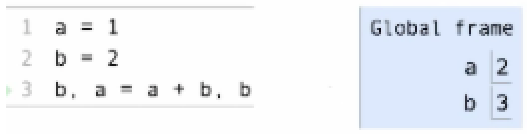
```
def adder(x,y):
    return x+y
f = adder(2)
```

2.5 Checkpoint 1 Solutions

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

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

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

   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 Booleans and Comparisons

Boolean's are specific variables that are set to the value of either **True** or **False**. However, every object in Python has a non explicit Truth value.

1. **False Values are the following**
   - None
   - False
   - 0
   - An empty string - ""
   - A few others that we will go over later on
2. **Truth Values are the following**
   - Any number besides 0
   - True
   - Any non-empty String
   - A few others that we will go over later on

2.7 Comparisons

In Python, we can check use a few operators to check if 2 values are equal. The first one we will go over is "==". This operator returns True if 2 values are equal and False otherwise.

```
>>> a = 1
>>> b = 1
>>> c = 2
>>> a == b
True
>>> a == c
False
```

In addition to equality, we can also check for inequality using the "!=" operator. In contrast, this operator returns True if 2 values are not equal and False otherwise.

```
>>> a != c
True
>>> a != b
False
```

The above operators check for equality and inequality for all the types that we have gone over so far; however, there are a few important operators that can be used to compare numerical values. These includes > (greater than), < (less than), >= (greater than or equal to), and <= (less than or equal to).

```
>>> d = 5
>>> e = 3
>>> f = 5
>>> f > d
False
>>> e < d
True
>>> f>=d
True
>>> d > e
True
```

2.8 Conditionals

Sometimes we would want to do some action only if some condition is fulfilled. In terms of this section, we want to perform an action if some condition is evaluated to True. We know that every item in Python can be evaluated to either True or False. So using a tool called **Conditionals**, we can evaluate only certain expressions. There are three conditionals in Python. The particular time when one can be used will be listed directly underneath.

1. **if**
   - Can use at any time.
   - We check if

   ```
   if some condition:
           some body
   ```

   is true. If so, we will evaluate the body following that clause.

2. **elif**
   - Can only use the statement if there was or another elif statement as the conditional before it.
   - If no conditional (if or elif) had a condition that evaluated to True and the condition evaluates to True for this one evaluate the body. Looks as follows

   ```
   if some condition:
           some body
   ...
   elif some other condition
           some other body
   ```

   is true. If so, we will evaluate the body following that clause.

3. **else**
   - Use as the closing conditional. Can only be used if an if statement was somewhere above it. It can follow elif statements as well; however, it cannot follow another else statement.
   - If none of the above conditionals had a condition that evaluated to True then just run the body of the else statement.
   - Basically the else statement should be thought of as a net that catches any case that was not accounted for.

Now that we have that out of the way we will go over a few examples of using conditionals.

1. First, we will look at the case with only an if statement.

   ```
   def superfan(x):
           if x >= 5:
                   return "It's a bit chilly"
           return "It's warm"
   superfan(10)
   ```

   In the above case, we run superfan with an argument of 10. The next step s then to check the clause $x >= 5$ and see if it is valid. In this case it is so then we return the String "It's a bit chilly". It is important to remember that after a return statement, **the function is done executing**. In this case, when we reach a return statement, the function call superfan(10) is equivalent to "It's a bit chilly". That means we do not continue to the line that returns "It's warm".

   If we, for example, had the function call of superfan(2), we would look at the if clause and see that $x >= 5$ is not true so we would not go inside of the body of the conditional. We would then continue past the body and return "It's warm".

2. Next we will look at the case where we have multiple if statements and elif statement(s)

```
def boom(x):
        if(x <=5):
                x +=1
        if(x >=6):
                x -=4
        elif(x <= 5):
                x+=10
        return x
boom(4)
```

   We run the function call boom(4). In the first step of the function, x is equal to 4. We then go to the first clause and check the condition, $x <= 5$. It is so we go inside of the body and set 2 to $x + 1$, resulting in x equaling 6. We reach another if statement- this implies that we are finished with the previous "if" series. We check the clause $x >= 7$ and it is not so we move to the elif statement and see if x ¡= 5 then we move to the body. It is so we set x equal to $x + 10$, or 15. We then return x which is 15. Note that the elif corresponds to the closest if statement so it does not matter that $x <= 5$ evaluates to True. Since the closest if statement evaluated to false, and there were no other elif statements, we evaluate the clause. However, if $x >= 6$ had evaluated to true we would not have evaluated the clause $x <= 5$.

3. We will look at a final case which will just be some configuration of if, elif, and else statements.

2.9 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.

```
def adder(x,y):
    return x+y
def square(z):
    return z*z
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:

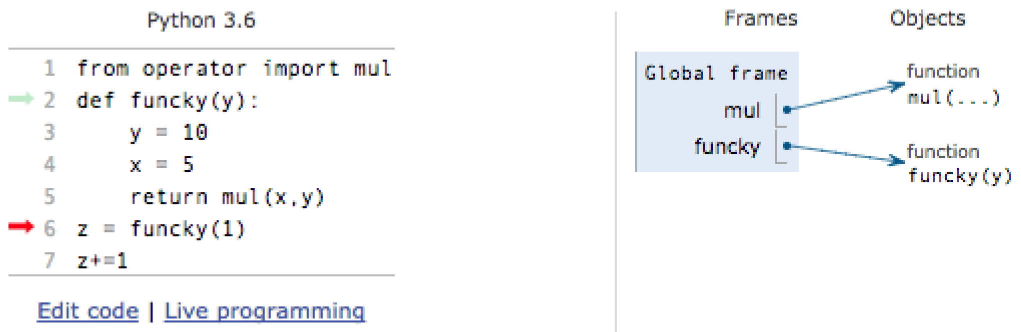
```
def adder(x,y):
    return x+y
def square(z):
    return z*z
a = adder(4,9)
```

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.

2.10 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.

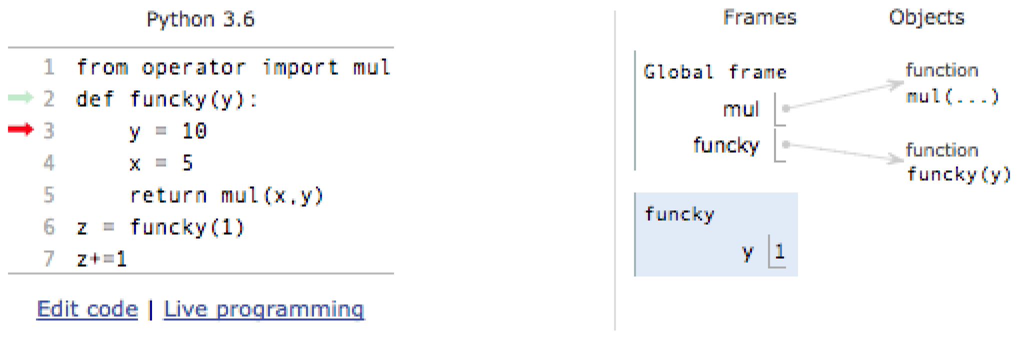
So far, we have dealt with somewhat basic functions and calls. With many function calls and variables, it becomes imperative that we have some organized method of keeping account of them. We will be using a visualization called **Environment Diagrams** to keep track of bindings, the association of a 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.



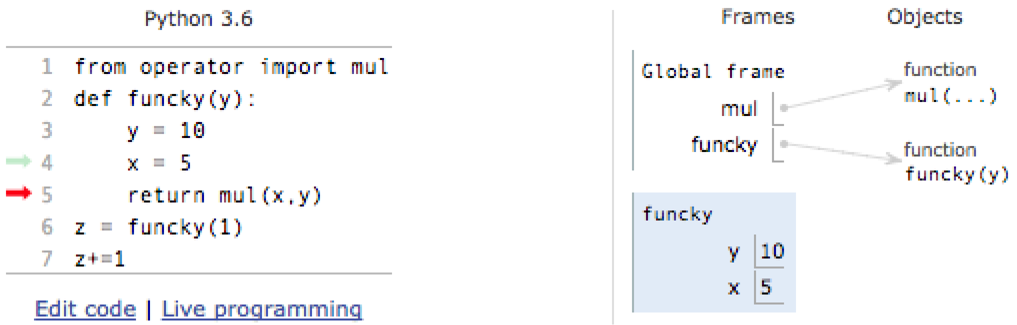
```
from operator import mul
def funcky(y):
    y = 10
    x = 5
    return mul(x,y)
z = funcky(1)
z+=1
```

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.

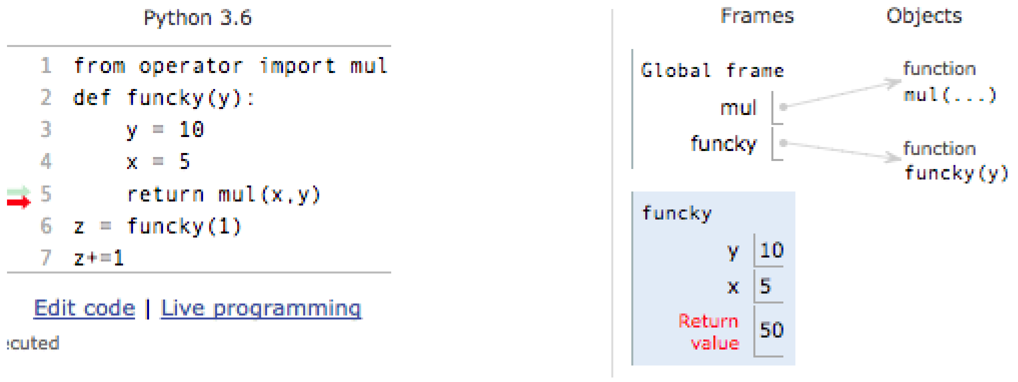


```
Python 3.6
1  from operator import mul
2  def funcky(y):
3      y = 10
4      x = 5
5      return mul(x,y)
6  z = funcky(1)
7  z+=1
```



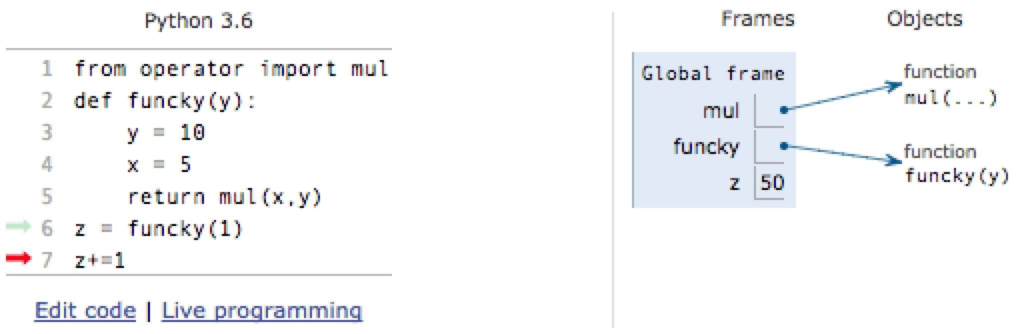
```
Python 3.6
1  from operator import mul
2  def funcky(y):
3      y = 10
4      x = 5
5      return mul(x,y)
6  z = funcky(1)
7  z+=1
```

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.

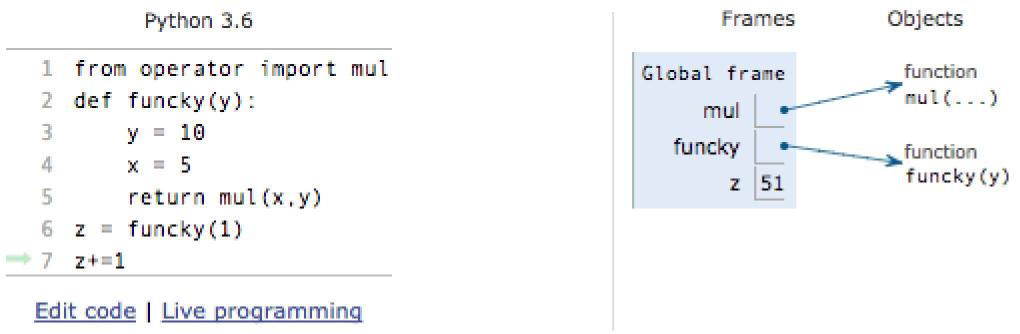


```
Python 3.6
1  from operator import mul
2  def funcky(y):
3      y = 10
4      x = 5
5      return mul(x,y)
6  z = funcky(1)
7  z+=1
```

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.



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



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

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:

```
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.

2.10.1 Nested Functions

Prior to this section, when we dealt with nested expressions they were just nested function calls in the following manner: func1(func2(....)). This is okay for certain cases but there are some draw backs. One of the main reasons we may want to use a nested function is because sometimes we want to only allow a function run for another function. In our old way of having nested expressions, we would need to have all the functions available to everyone.

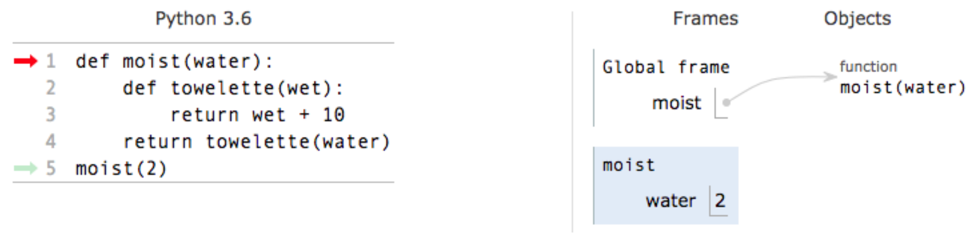
Below is an example of a nested function:

```
def moist(water):
    def towelette(wet):
        return wet + 10
    return towelette(water)
```

We have a function moist that takes in a parameter water. Inside the function moist there is another function towelette which takes in a variable water. Say we were to run the following line of code:

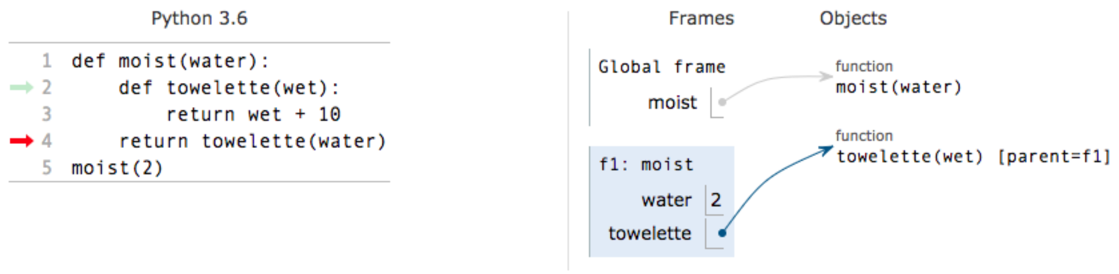
```
>>> moist(2)
```

What would we get as a return value to this function? Well it can get a bit weird because we would return a function call when we run moist(2) so let's walk through it to avoid any confusion:



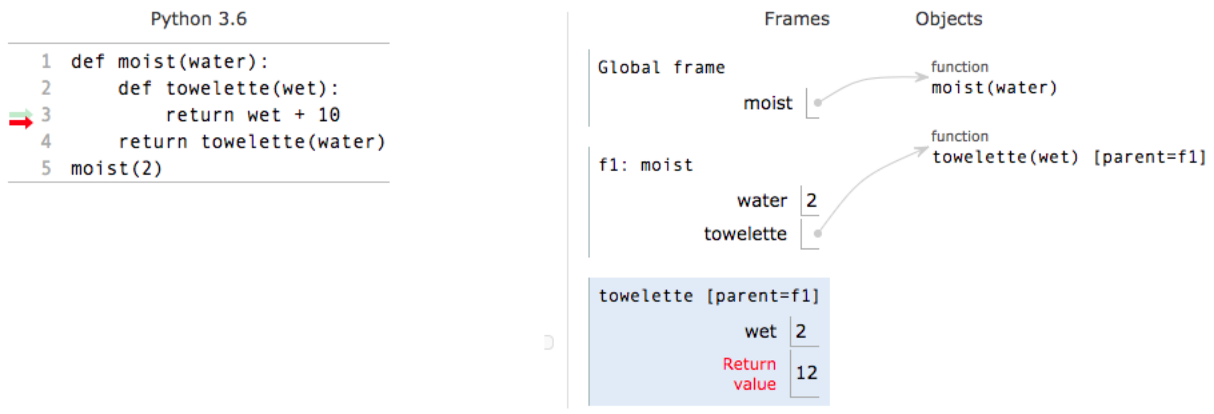
```
def moist(water):
    def towelette(wet):
        return wet + 10
    return towelette(water)
moist(2)
```

We start by doing a normal function call to moist(2). The parameter water is set to 2.



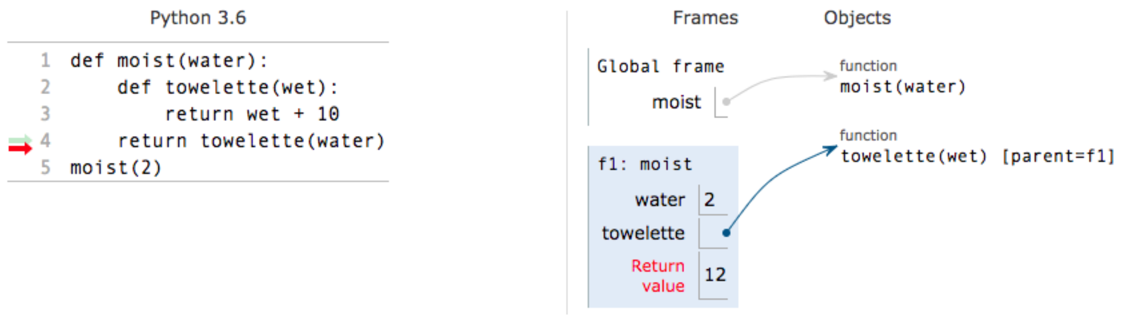
```
def moist(water):
    def towelette(wet):
        return wet + 10
    return towelette(water)
moist(2)
```

Now we get into the more interesting part. We now go to the return statement in moist and return towelette water. Now going back to our earlier rules, since we are returning a function call and not a function, we evaluate this statement further. we know that we evaluate this function before finally returning a value so let's go a bit further!



```
def moist(water):
    def towelette(wet):
        return wet + 10
    return towelette(water)
moist(2)
```

We go into the function towelette with wet assigned to 2. We then return wet + 10 which is 12, making the return value for this frame equal to 12.



```
def moist(water):
    def towelette(wet):
        return wet + 10
    return towelette(water)
moist(2)
```

We have returned out of the frame for towelette, but we aren't done yet! Remember, that return value was for a function call that itself was in a return statement! So we return out of the frame for the function moist with a value equal to the return value of towelette(water), or 12.

When drawing out the function calls for nested functions remember to keep your environment diagrams a bit clean, and to label the parents of frames, otherwise things cant get a bit crazy.

2.10.2 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.

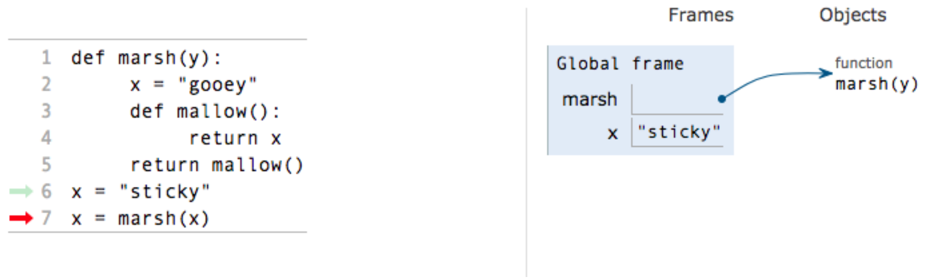
2. Child frames can view variables in parent frames; however, they cannot directly edit those values. There will be exceptions to this which we will go on later.

It is important to note that since functions are essentially variables- just a name defined to some set of operations, these same rules also apply to functions.

We will over a basic example now showing some of the ways the scope of variables can be made a bit tricky.

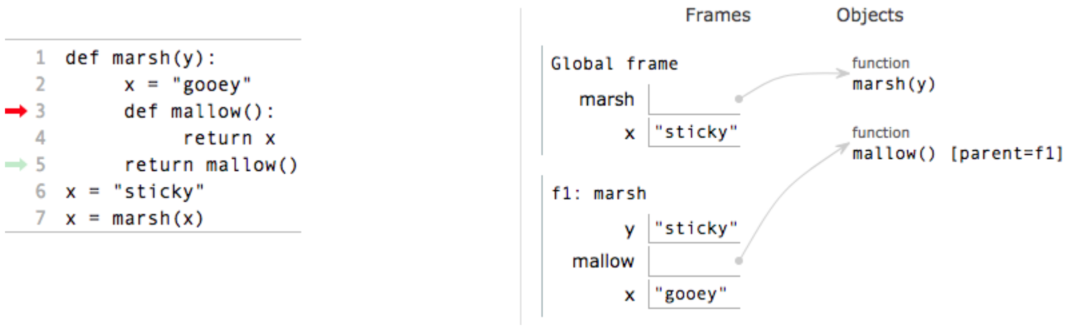
```
def marsh(y):
      x = "gooey"
      def mallow():
            return x
      return mallow()
x = "sticky"
x = marsh(x)
```

Now there sure are a lot of "x"'s on here so it can be a bit confusing to see which x goes where. So let's do this step by step.



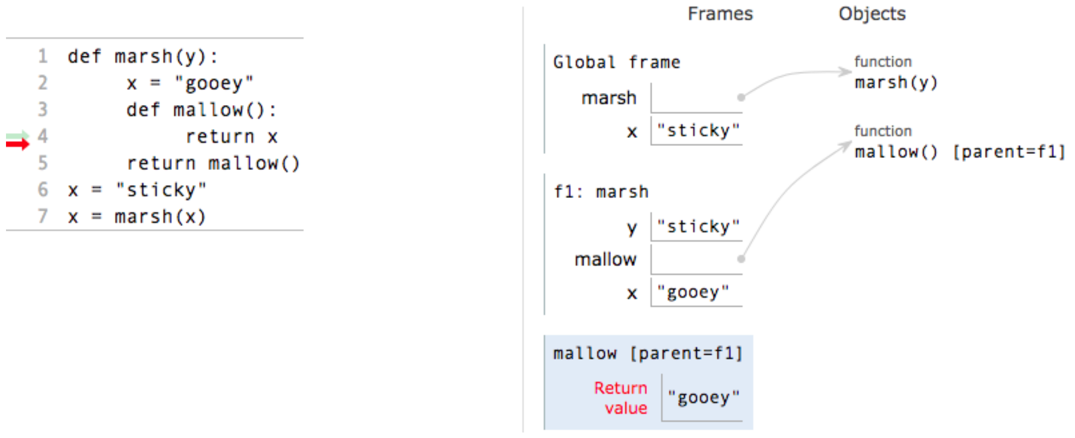
```
def marsh(y):
      x = "gooey"
      def mallow():
            return x
      return mallow()
x = "sticky"
x = marsh(x)
```

We start off with x equal to the string "sticky". We then set x equal to the value of marsh(x). However, we have not yet found the value for this yet, so we will first evaluate the function call marsh(x) where x is the string "sticky".



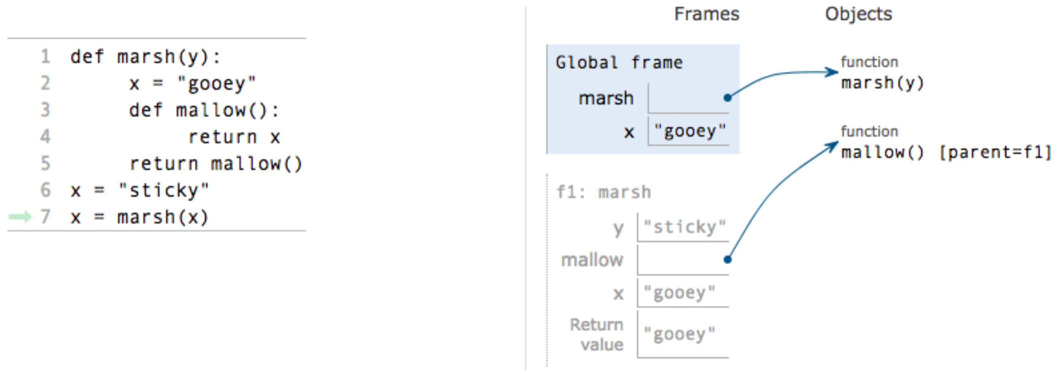
```
def marsh(y):
      x = "gooey"
      def mallow():
            return x
      return mallow()
x = "sticky"
x = marsh(x)
```

Now we go into function marsh and the parameter y is set to the passed in value, which was "sticky", or x in the global frame. Then the x value in this frame is set equal to gooey. After this statement, we have the line that returns mallow(). We must now evaluate the function call for mallow().



```
def marsh(y):
     x = "gooey"
     def mallow():
          return x
     return mallow()
x = "sticky"
x = marsh(x)
```

We go into mallow and then we have a new dilemma- we are returning x; however, there is no x in this frame. Well according to our rules, we can view the values in our parent frames, but we cannot modify them. Since we are not redefining x, we can get the value of x from our parent frame, in this case f1, or our call to mash(x). The value of x in this frame is the string "gooey", so we return "gooey".



```
def marsh(y):
     x = "gooey"
     def mallow():
          return x
     return mallow()
x = "sticky"
x = marsh(x)
```

We then return mallow() in f1 which is equal to "gooey". After this, we finally have the value of marsh(x)- "gooey". At this point we redefine x to be equal to "gooey" in the global frame.

Chapter 3

Functions revisited

3.1 Iteration

We have dealt with simple functions; however, the power of functions can be displayed through a concept called **iteration**. Iteration is a way with dealing with repetition within a function. For example say you want to find the sum of the first n numbers. We would want to iterate over each number and keep track of a sum, and for each number, we would add it to the sum.

We can use the **while** statement to help us loop over elements. While loops take the following format:

```
while some condition:
        statements that occur  is the condition is true
(optional) statements that occur after the condition is evaluated to false
```

The statement that occurs in the while block will occur over and over until the statement becomes false. This usually means that at each step of the while loop, we want to somehow get closer to terminating the loop- by enumeration or some other method.

. In the prior example, we stated that we wanted to sum over the first n numbers. Our condition in this case would be while our current number is not greater than n and at each step in the while loop, we would want to increment our sum by the number we are currently on. More concretely:

```
1 def sumofN(n):
2         i = 0
3         sum = 0
4         while i < n:
5                 i = i + 1
6                 sum = sum +  i
7         return sum
```

To step through this code press **here**. You can change the arguments to any number over 0 and get a feel for what's going on.

It is important to realize that on line 6 we evaluate i+1 and then set that equal to i. Similarly we rebind the variable sum to the sum of the variable sum and i.

When dealing with loops, it can be easy to fall in the trap of ”off by one errors”. These are basically errors that result when you overcount or undercount your expected terminating condition. For example, in the abovecode consider what would happen if we set the while loop to i <= n instead of i <n.

Additionally, it is possible that your while loop never terminates. If you never get any closer to a terminating condition, your loop will go on forever. This can happen in the following case”

```
        int i = 1
        while i > 0 :
                i = i+1
```

Since i is always increasing, it will never approach 0 and as a result, it will never be able to exit from the loop. If you ever get stuck in an infinite loop, press control c on your computer and you can exit from the loop.

3.2 A Deeper Look at Higher Order Functions

We talked about nested functions earlier- now we will go more in depth and talk how to utilize them more. Below we have a standard higher order function:

```
def PB(x):
        def Jelly(x):
                x = x + 5
                return x
        x += 1
        y = Jelly(x+2)
        return x + y
```

Now look let's run PB with an argument of 5. Well start off like any other function and set x equal to 3 in the PB frame. We then set the variable Jelly equal to the function Jelly. Finally, we increment x by 1. Now we perform the function call Jelly(x+2).
The function is called with an argument of x+2. We are currently in the f1 frame so we make Jelly's argument of the value of x, 5, plus 2, or 7.

We are now in the f2 frame. f2 has a variable defined in its scope named x. We define x **in this frame** to be equal to 7. Note that it is still 5 in the global frame.

We then change x's value to be x + 5. Since x is 7 in this frame we change x to be equal to 12. Now we are done with the function and it returns 10.

We have now exited the function and returned to f1 We are now back in the global frame and we take the return value of our function, 10, and set the variable y equal to it. Finally we return x+y, which is 10+5 or 15.

3.3 Lambda Functions

We will now go over the one line function called the **lambda function**. We know we can write functions like this

```
def addFive(x):
        return x + 5
```

We can rewrite this function in the following syntax.

```
lambda x: x + 4
```

This one liner called the lambda function can be quite useful and takes the form of:

```
lambda parameter1, parameter2: some operations that you want to return
```

Lambda's are essentially functions that have no variable name associated with them. Typically we want to use lambdas if we want to do some quick operation or mapping and want to save it. It is important to realize that lambda functions are no different than other functions. For example, if we have:

```
>>> x = lambda x :  x  + 5
```

we can evaluate the function on the argument 6 by doing.

```
>>> x(6)
11
```

This is no different the same way we apply arguments on variables equal to normal functions. This means that whenever we create a lambda function we do no call it (even if there is no parameter- in this case it is just equivalent to a function with no parameter). This is a small point but a very important one.

3.4 Recursion

We know that we can call functions within other functions- this leads us to an interesting point: from one function we can call the same function. If a function calls itself we call that function **Recursive**.

If we keep calling a function on itself with the *same* argument, we know that we will get into infinite recursion- our process will never end! So how do we avoid this? Well we can change our function's argument so that it gets closer to some terminating condition, and eventually reaches it.

What is one process that we discussed earlier that has a terminating condition with some argument that continually gets smaller? The answer is While Loops! In fact, recursive function can usually be rephrased as while loops and vice versa. So why would we want to use one over the other? There are a few reasons but one of the main ones is that sometimes it can be more natural to write a function one way versus the other.

Say we want are given some number x and we want to find x!. How can we do this? Let's start off with iteration which may be more intuitive.

```
def factorial(x):
        sol = 1
        while x > 0:
                sol = sol * x
                x -= 1
        return sol
```

Let's now go through an example of running through the factorial function with the number 3. We will consider each iteration to be one "step". In the first step we multiply our current product (1) by 3. We then decrement x by 1, making it 2. We then multiply the current product (3) by 2 making it 6. Again we decrement x by 1 making it 1.We then multiply our current product (6) by 1 making it 6 and we decrement x to 0. Since x is 0 our while loop is terminated.

Now let's look at the pattern that this function yielded. We had 3*2*1=6. More generally, for some n, we had (n)*(n-1)*(n-2)...(1).

In this case, we perform our iteration while x is greater than 0. So what is the terminating case? Well we want the function to stop if x ever falls to some number less than or equal to 0 so we can have some conditional in our function that follows the format:

```
if x <= 0:
```

Additionally, we multiplied the value of x at 1 iteration with the value of x in the next iteration and so forth. This means that we should have the following line of code somewhere in our function:

```
x*factorial(x-1)
```

Now lets put it all together. If we were to call this on 1 we would immediately get to the terminating condition. What would we want to do? Well we need to return something for the function call factorial(0) otherwise we would be multiplying 1 by None. So we know we need to return something, but what is it? Well if we return just 0 then we get a problem since 1! is not 0. What do we do then? Well if x is less than or equal to 0 we can simply return 1. We then know our terminating condition or base case will be

```
if x <=0:
        return 1
```

Let's take the value for factorial 2. We know we will have a call 2*factorial(1) = 2*1*factorial(0). From here we can come up with our recursive function.

```
def factorial(x):
        if x <= 0:
                return 1
        return x*factorial(x-1)
```

To verify this is correct let's try it with the argument of 3. We have:

$$3 * factorial(2) = 3 * (2 * factorial(1)) = 3 * (2 * (1 * factorial(0))) = 3 * (2 * (1 * 1)) = 6$$

. We can verify that this is correct with a calculator or computation.

In general, coming up with a recursive function involves the following steps.

- **Base Cases** Find the base cases. In other words, see when you want your function to terminate
- **Argument Reduction** Find out what you want to do to your arguments to the function. Somehow get closer to your base case- usually done by some arithmetic.
- **The Recursive step** This step requires you to figure out what you want to do for each argument until the base case. In many recursive cases you want to do something with the combined Recursive steps (use your problems towards your base case to solve the original problem).

An optional first step is to write the function iteratively first and convert it to a recursive function; however, as you gain practice with recursion you will find this step to be a bit redundant.

3.4.1 A Quick Side Note on Recursion

The idea of Recursion can frequently be a bit hard to grasp so here will be a bit more of an intuitive explanation rather than a code based one.

Say that you are in a theater and it's really dark and you want to know how many rows are in front of you but you can only see 1 row in front of you. How can you figure out how many rows are in front of you?

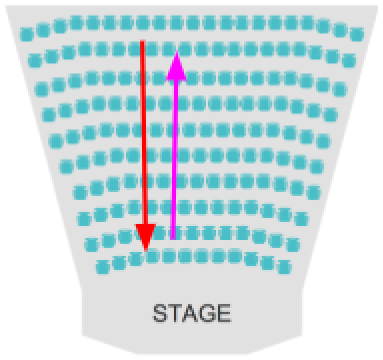
Well you can ask the person in front of you how many rows are in front of them and they can ask the person in front of them how many people are in front of them and so forth. This process continues until the person in the 0th row (the very front row) is asked this question. The person in the very front row would respond with "0" since there are 0 rows in front of them. The person behind them then gets that result and adds 1 to it, since there's the 0th row person didn't include themselves in the count, and then tells the person behind them the number. This process continues until we reach the original person.

Recursively this would look something like the following function

```
def theater(x):
        if x is in the first row:
                return 0
        return 1+ theater(person in the next row)
```

In this case reaching a person in the first row is the **base case**. Asking the person in front of you how many people are in front of them is the **argument reduction**. Adding up 1 at each argument is the **recursive step**.

Visually this looks like this. We go all the way down to the front of the theater then each person tells the person behind them how many people are in front of them.



3.5 Tree Recursion

There are a special class of recursive functions that are called **Tree Recursive**. These are functions that call themselves more than once per call. Such a function may look like the following:

```
def tree_recurs(n):
        if n == 1:
                return 1
        return tree_recurs(n-1)+tree_recurs(n-1)
```

Tree Recursion is given its name because each function call would have more than 1 function call come out of it and "branch out". Tree recursion c an be tricky and it can be useful to write out the recursive relations that we did earlier for factorial.

Chapter 4

Built in Data types

4.1 An Introduction to Lists

We have gone over how variables can be assigned to one value, but let's go up one more step. Now, we will assign variables to sequences of values. The first object that we will go over that fits this criteria is called the **list**.

Lists, as the name implies, store a list of values. For example. The way that we instantiate a list is as follows:

```
>>> lst = [1, 2, 10, "Kettle Corn"]
>>> lst
[1, 2, 10, "Kettle Corn"]
```

4.1.1 Using Lists

We can also interact with lists in a few ways that make them very natural to use when manipulating data. For example, we can get the 1st element of our previously created list by performing the following call:

```
>>> lst[1]
2
```

A somewhat tricky part about this syntax is that the 1st element in the list occurs after the 0th- that is, our list is zero indexed.

Another useful function that Python has provided us to deal with lists is the *len* function. If we call len on a list, we will get the length of the list.

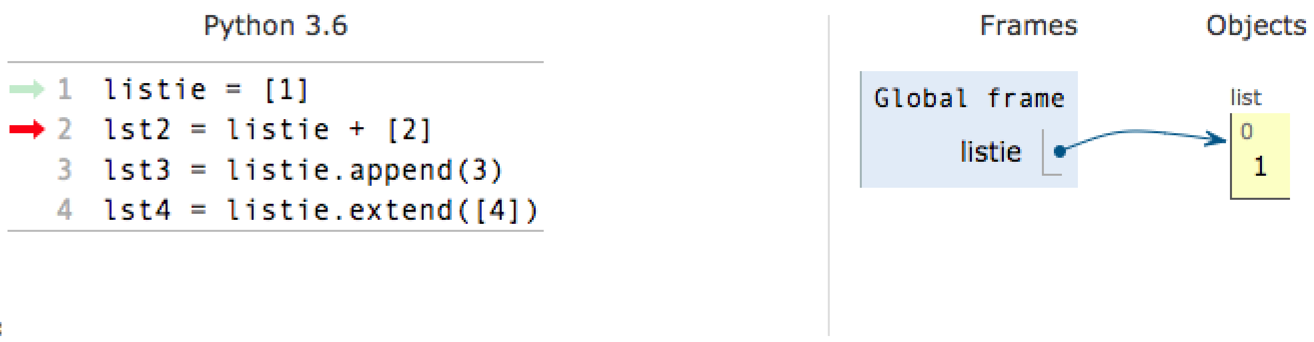
```
>>> len(lst)
4
```

Now what values can we store in a list? Well just about any. We can store numbers, Strings, and even other lists. Storing lists inside lists may seem a bit strange, but it can be useful. For example, we would use this is we wanted to store the longitude and latitude of the 5 nearest airports. Below is the syntax for putting a list inside of a list:

```
>>> coords = [[102.355, 101.111], [54, 32]]
>>> coords
[[102.355, 101.111], [54, 32]]
>>> coords[1]
[54,32]
>>> coords[1][0]
54
```

4.1.2 Adding to Lists

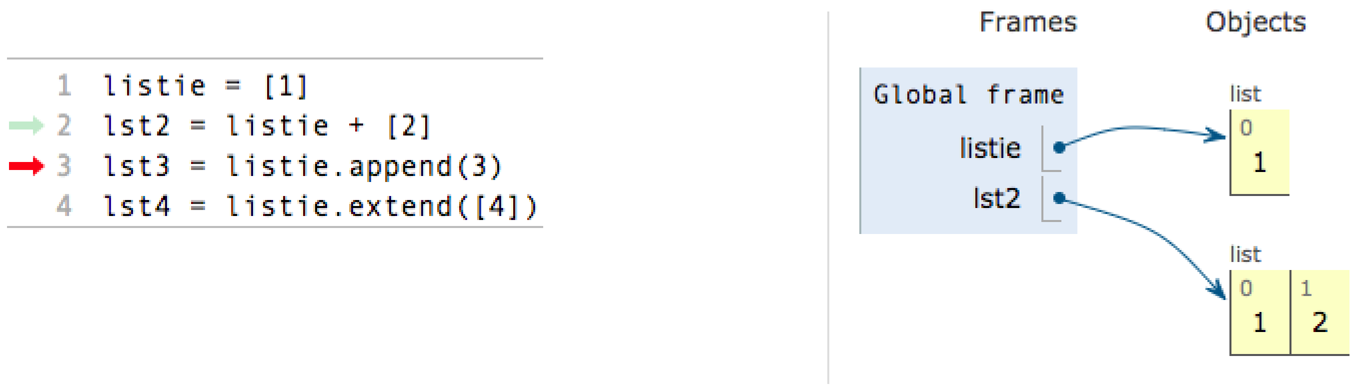
Sometimes we want to change our list of data in some manner. The first modification that we will go over is adding to a list. There are three canonical ways to do this: the "+" operation, ".append", and ".extend". Each of these methods has their own syntax and unique attributes.



```
listie = [1]
lst2 = listie + [2]
lst3 = listie.append(3)
lst4 = listie.extend([4])
```

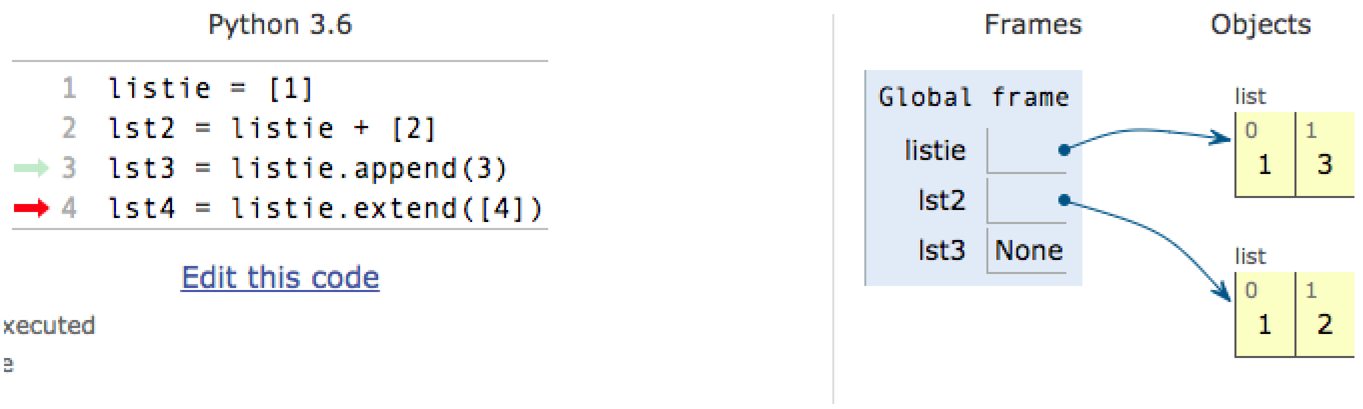
We will use the above block of code to illustrate the differences between adding to a list. The PythonTutor link can be found **here**.

The "+" operator adds an arbitrary amount of lists and returns a new list. The outcome of a "+" operation is shown below. It is extremely important to realize that when we perform the "+" operation on lists, **the original list does not change**. This can be seen above where listie is still [1], but the new list, lst2, has the value of [1,2]



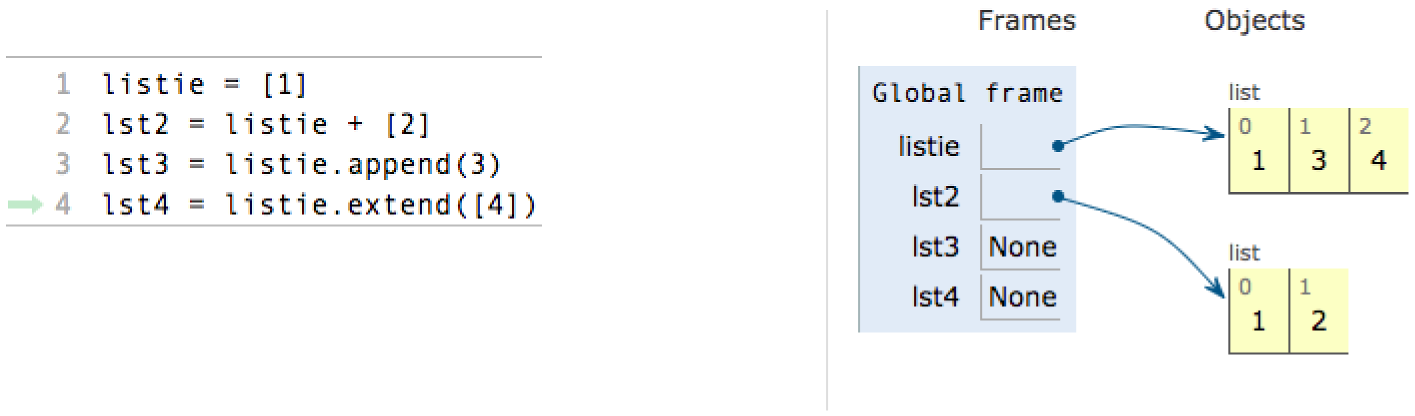
```
listie = [1]
lst2 = listie + [2]
lst3 = listie.append(3)
lst4 = listie.extend([4])
```

Alternatively we can use the ".append" method. This method takes in 1 value and adds it to the end of the list. For the same list as above, let's perform the ".append" method.
Unlike the "+" operator, **the ".append" method does not return a new list, rather it modifies the original list**. In the above visual, we can see that lstie has been modified to include the number 3 at the end and lst2's value is None. The reason why lst2's value is None is because the .append function has no return value.



```
listie = [1]
lst2 = listie + [2]
lst3 = listie.append(3)
lst4 = listie.extend([4])
```

Finally let's look at the ".extend" syntax. The ".extend" function takes in a list of arbitrary length and adds it to the end of the list

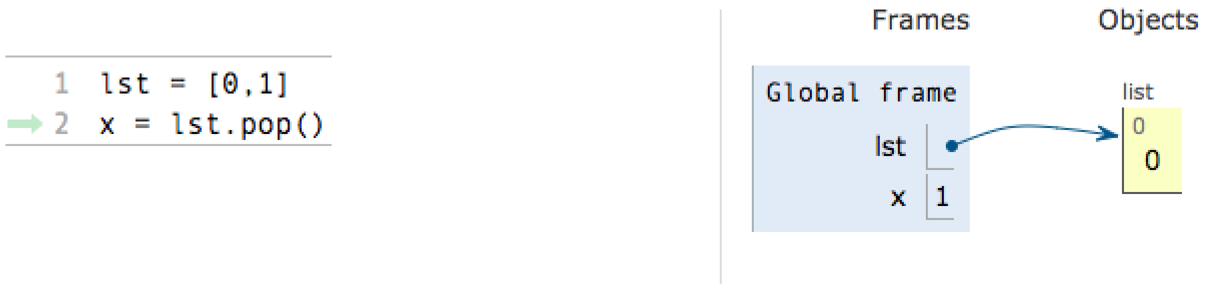


```
listie = [1]
lst2 = listie + [2]
lst3 = listie.append(3)
lst4 = listie.extend([4])
```

Just like append, extend modifies the actual list, it does not return a new list like the "+" operator. The list listie has been modified to include 4 in this step and lst3's value is None for the same reasons that .append follows this pattern.

4.1.3 Removing from Lists

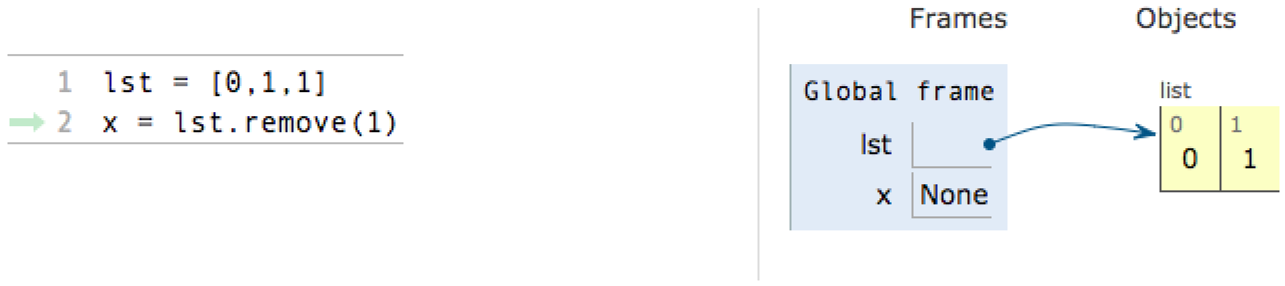
Removing from a list is a bit more straightforward. We will go over two ways to remove from a list. The first one involves the pop syntax. The pop function has an optional index argument in which the user can input the index that they want to be removed. If no argument is specified, the last element is removed. The return value is the element that is removed.



```
lst = [0,1]
x = lst.pop()
```

Note how when we pop the original list is affected.

We will now use the remove syntax. The remove function takes in the value element that you want to remove and removes the first instance of it.



```
lst = [0,1,1]
x = lst.remove(1)
```

The original list is affected and the first instance of the number one is removed.

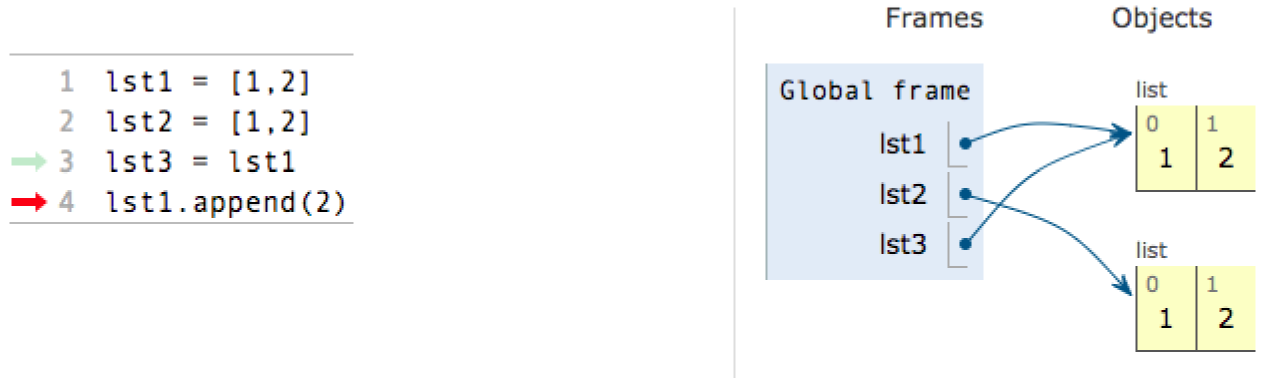
4.2 Advanced Lists

4.2.1 Equal Lists

Earlier in this book, we talked about certain types. These types are called primitives and include integers, characters, booleans, and strings. However, lists are part of a different type- Objects. We will go over Objects in more depth in the next chapter but for now we will just discuss this in terms of lists.

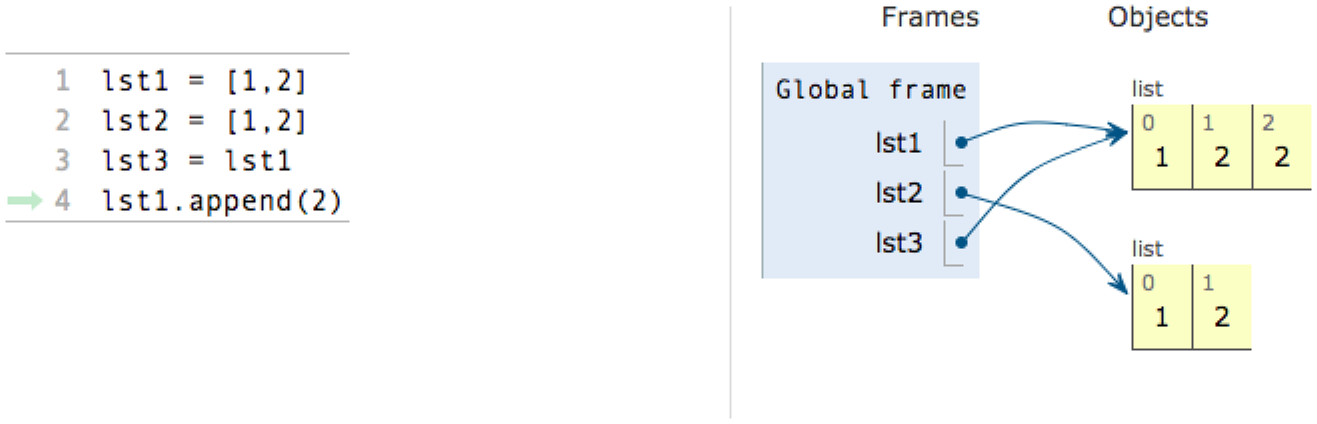
Two variables whose values are lists are said to be equal if they "point" to the same list. By this I mean that two variables are equal if they both point to the same container (thinking of lists as containers for values).

Even if two lists have the exact same values, it does not necessarily mean that the lists are equal. This can be visualized **in the following example**. Note how lst1 and lst2 are not pointing to the same list even though they have the same values at first (1 and 2). On the other hand, lst1 and lst3 point to the same values container because lst3 was set equal to the container that lst1 was pointing to.



```
lst1 = [1,2]
lst2 = [1,2]
lst3 = lst1
lst1.append(2)
```

Also note how after the append, the values in the lists lst1 and lst3 change because they point to the same container.



```
lst1 = [1,2]
lst2 = [1,2]
lst3 = lst1
lst1.append(2)
```

4.2.2 Slicing and Other

4.3 Tuples

Tuples are just like lists except they are immutable. In other words, once a tuple is initialized it cannot be changed. Unlike lists, we initialize tuples with parentheses rather than square brackets. Tuples can be indexed in the same way as lists; however, tuple indices can never be reassigned, nor can anything be added or removed.

4.4 Dictionaries

Sometimes, we want to store key value pairs- in other words, we want to associate some key to a property. An example of where this method would be useful is associating students to their grades in a particular class. The name of the student would be a String and the grade would be an integer. An important thing to note is that each student is unique but their grades may not be. For example, John and Jim could both have 90% in the class. Additionally we may want to change someone's grade from a 90 to 95.

The above observations help us form the intuition for a ***Dictionary***. Dictionaries are made up of 2 components, an immutable key (a key that cannot change) and a value that can change. Additionally, key's must be unique. That is if you insert 2 keys into a dictionary, the more recent one's value will be the value in the dictionary. A side note about dictionaries is that they are inherently unordered- instead of get a value by index, we get a value by its key. **insert example here**

4.5 Iteration over Lists

Lists are indexible and this means that they are quite natural structures to iterate over. To iterate over lists we can use either while loops or for loops. We will look over how we can return the elements in a list using different

```
lst = [1,2,3]
counter = 0
sum = 0
while counter < lst.length:
        sum = sum + lst[counter]
        counter+=1
```

While this is fine, the for loop can be more concise. We can use the in range syntax to iterate over every number from 0 to the number (inclusive at the 0 and exclusive at the number). Note that range has an optional start parameter, but by default it is set to 0.

```
lst = [1,2,3]
sum = 0
for i in range(lst.length):
        sum = sum + lst[i]
```

Finally we can just add all the elements in the list without even indexing. Instead we can use the in syntax to get each element from the list.

```
lst = [1,2,3]
sum = 0
for i in lst:
        sum = sum + i
```

4.6 Comprehensions

Sometimes, it can be useful to condense a for loop over a list structure. This is where comprehensions can come in handy. Comprehensions allow us to generate a new list structure by looping over some set other list or tuple. An example of a list comprehension is in the following we would return a list with every number in the previous list incremented by 1.

```
lst = [1,2,3]
[i + 1 for i in lst]
```

An easy way to think about this is as a for loop that is iterating over a list and creating a new list for each element in the list. There are a few complicated operations that you can do with list comprehensions.

One such example is the conditional statement that goes within list comprehensions. This takes the following form.

```
[i + 1 if i < 10 else i + 2 for i in lst]
```

Now you cannot have elif statements in list comprehensions but you can have nested conditionals like in the following:

```
[i + 1 if i < 10 else if i >20 i + 2 else i+3 for i in lst]
```

While list comprehensions do reduce the amount of code that needs to be written it can sometimes be at the cost of readability. As a programmer it is important that you not only make your code for yourself but others who may stumble upon it.

Chapter 5

Abstracton and Classes

In the last chapter we went over lists. Now we will go over other mechanisms to store data.

5.1 Basic Classes

5.2 The LinkedList

5.3 Trees

5.4 Recursion over Trees