UDACITY Fundamentals of Programming

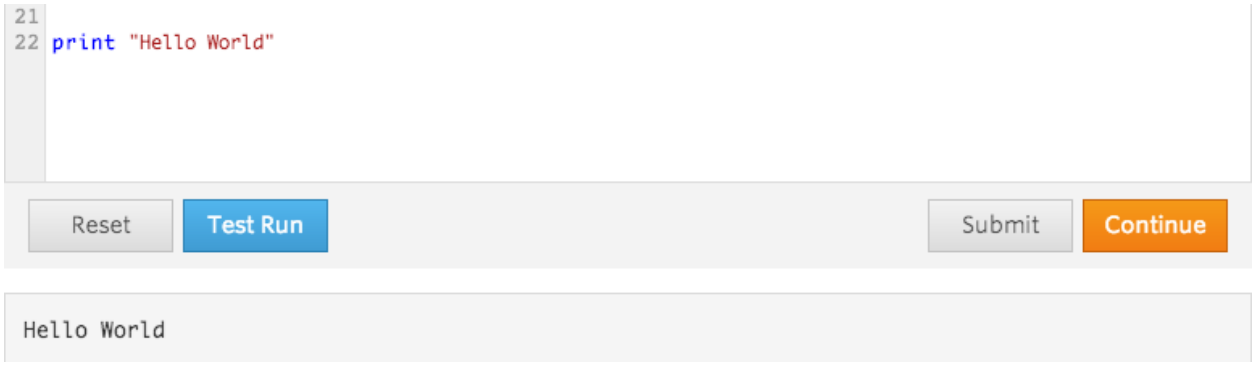
If Statements, For Loops, Functions

Table of Contents

In this module, we will be reviewing three of the most critical fundamental concepts of programming in the Python language: types of conditionals, loops, and functions. Please check out the in-browser IDE (fancy word for a place that runs Python code) provided to you so that you can experiment and write code as we go through this document. For those unfamiliar with the language, don't worry! This first module will introduce you to the Python programming language and help you get comfortable with it. Now, let's start off by saying Hello to the world in our program!

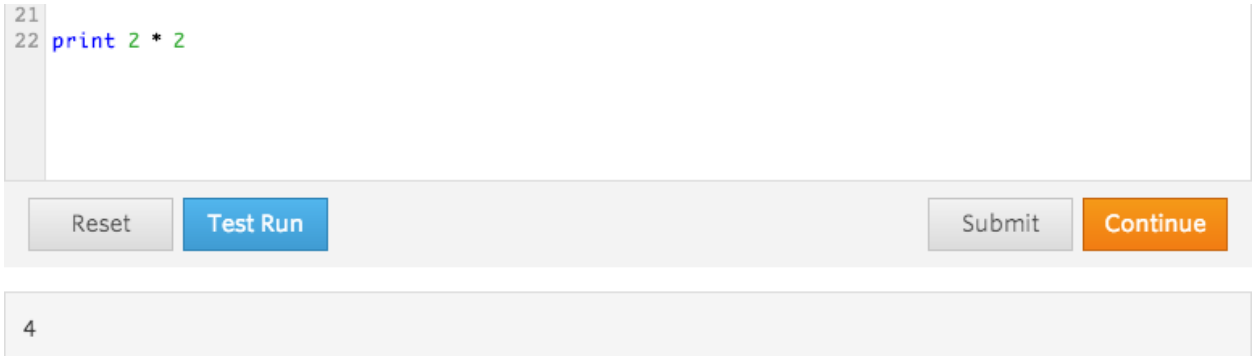
Hello World

In programming languages we can input commands that the machine will execute. For example, if we want to say Hello World in Python, we would write:
print "Hello World"
Then press the Test Run button and see below:



Here, my program tells the machine to print the sentence "Hello World," which it then prints in grey text.

We could also solve arithmetic problems and print out answers. For example, we can write code to see that 2 * 2 = 4.



We can also solve more complex problems and see the result. For example, what is 2^{10}? We can see the result of this by typing print 2**10, another way of saying 2^{10}.

```
21
22 print 2**10
```

Reset | Test Run | Submit | Continue

```
1024
```

We can also save information inside of variables so we can change or use it later on. For example, here I get the value of 2^7, save it in variable x, and afterwards add 500 to it.

```
21
22 x = 2**7
23 x = x + 500
24 print x
```

Reset | Test Run | Submit | Continue

```
628
```

This can get pretty confusing. I'll go ahead and add what are called comments. Comments are denoted with a # and allow programmers to add some comments or remarks about the code. This information does not affect our commands in any way, they are just there so that other programmers can understand the intentions of the code.

```
20
21 #This is a comment!  I stored 2^7 into x
22 x = 2 ** 7
23 #I added 500 to x
24 x = x + 500
25 #I then print out the value of x
26 print x
```

Reset | Test Run | Submit | Continue

```
628
```

To learn more about variables, check out this video segment from the Introduction to Computer Science by Professor Dave Evans!

Types of Variables

This will be a more tedious section, where we'll go over some vocabulary and make some distinct points that will help guide us through the rest of this document. While we'll often work with numbers, there's also a strong need for us to work with words and other forms of information, which we will refer to as data types. One example of such a data type we've already seen is the String, which we saw in the form of "Hello World!" Other data types that computer scientists work with include the boolean and the list.

Integers and Floats

When dealing with numbers, we can break them down into two basic variables types: (there are actually a large variety of them but we only need to worry about 2 of them).

- Integer - A whole number that can be positive or negative (including zero)
    - Examples: 0, 2, 1337, 50012, -1492
- Float - A decimal number
    - Examples: 1.324, 1.0, 0.0, -1.35353

It's usually enough to refer to these two data types as numbers. We will rarely need to worry about the difference between these two types when working with Python.

String

The String is a fancy way of saying that we're dealing with a sequence of letters, numbers, and other special characters (e.g. $, %, #,). If you want to see what is accepted as a character check out the ascii table of characters. As a word of warning though, this table will look very complicated. The only column you need to concern yourself with is the "char" table for now. To write a String in Python, we wrap it with either single quotations or double quotations. Here are some examples below.

- "September"
- "Blarg, this sentencedoesn't;;makesense!!!"
- 'This is a single quotation String'
- """This is a multi
  line String! It can take up as many lines
  as it wants because I used three quotations!"""

Here are examples of what will not be Strings.

- "This is not a complete String because I started the String with a double quotation and ended with a single quotation, so it never finished'
- ""I did two quotations, so I made a String with nothing beforehand and with nothing after""

Boolean

The Boolean can be closely compared to a light switch. Like how the average lightswitch will have two states - an on and off state - the Boolean has two states. This binary decision is usually expressed with a True and a False, True being 1 and False being 0. In this section I will not be going into the technicalities of the Boolean, so it will be enough to think of the boolean as being either True or False. Below I will share some examples of how we might write booleans.

- x = True #Set the variable x equal to the True value
- x = False #Set x equal to False value

Keep in mind though that the Boolean differs in that it is a statement: a matter of fact. We can ask questions in real life and see what the results are: "Is x equal to 5?" → True, it is!" We can do the same in programming to get information that helps us function. To check if x is equal to 5, we would go ahead and write this:

- x = 5 #Set the variable x equal to 5
  print x == 5

Here, we're using something called a relational operator (More on this in the next section) to establish the relationship between the right hand side and the left hand side of the statement. Is x equal to 5? In this case, we would see that the result is true. Here are more examples that demonstrate how we might use relational operators (specifically is equal to) to interpret information.

- y = 5
  x = (y == 5)
  print x
  - Here I check if y is equal to 5. To check equality, we use ==. Because y is equal to 5, I have set x = True in a roundabout way.
- y = 4
  x = (y == 5)
  print x
  - Check if y is equal to 5. Because y is not equal to 5, I have set x = False here. I think print the value to see what it turns out being.

Relational Operators

Relational Operators allow us to effectively turn our statements into simple Boolean values that we can use. We've already seen a very common operator, which checks equality. The == operator will check whether the left value holds equal value to the right value, and will return True or False depending on the result of the comparison. Here is a list of some example operators that we can use and what it means.

- == #Checks Equality
- < #Check if the left value is less than the right value
- <= #Check if the left value is less than or equal to the right value
- > #Check if the left value is greater than the right value
- >= #Check if left value is greater than or equal to right value
- != #Check if the two values are not equal to each other

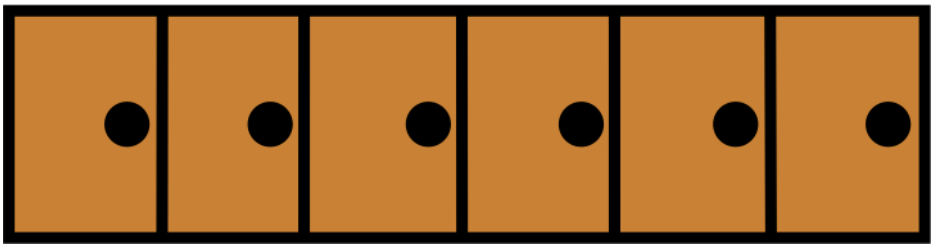
Booleans can take a lot of time to get used to. I would recommend going to your programming playground within Udacity and giving things a try and seeing what happens!

Lists

When dealing with a lot of information, there's a strong need for us to be able to store a lot of variables easily. For example, maybe we're doing a little statistics and getting the mean of a list of numbers. The List helps us organize and store a seemingly limitless amount of information into a single container. We can represent a list by placing values inside of brackets like such:

- my_list = [1, 2, 3, 4, 5]
- string_list = ["apple", "orange", "banana"]

When creating a list, we can imagine that we've created a series of compartments like we would see at the post office.



We can then insert elements into each compartment, take a look at those elements, or replace elements at will. We can then just as easily pull the elements out for analysis. We do so as shown below.

```
22 my_list = ["apple", "orange", "banana"]
23 print my_list
24 print my_list[0]
25 my_list[1] = "mango"
26 print my_list[1]
27 print my_list
```

Reset | Test Run

```
['apple', 'orange', 'banana']
apple
mango
['apple', 'mango', 'banana']
```

As we can see, we started off with a list that contains three elements. I started off by printing the list as a whole, so I got back my apple, orange, and banana. Afterwards, I checked what is in the first element of my list (more on this later). I then chose to go to my second element, my_list[1] which has "orange", and then replaced that value with "mango". Finally, I went ahead and printed out the second

element to see the change I made and then I printed out the list as a whole to see what happened there.

At this point, what should stand out to you is how we referenced elements in our list. When we wanted to get the first element in our list, we did my_list[0]. Oddly, we stated that our first element is at position 0 while the second element is at position 1. This is a critical detail in Computer Science that is important to remember. In general, computer scientists like to index starting from 0. So when we reference position 1, it's actually going to be 0 for us. So practically, it'll be a lot easier if we reference "apple" as being our 0th element in our list. You can see a somewhat technical discussion on the topic here. This blogpost also provides an interesting perspective.

Lists also have special properties that we can take advantage of. If we want to determine the length of a list, then we can do so with the len() command. Here's an example of how we would use it:

- len(my_list)

We can also add elements, or lockers, to our already made list. To do so, we'll simply append. Here is an example of what our line of code will look like:

- my_list.append("pineapple")

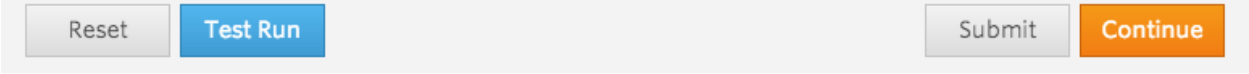
After this command, we'll find that the element pineapple will be added to the end of our list, so the length of our list will be one longer as well.

Conditionals

We'll quickly find that things can feel very constrained using only simple statements. Imagine if we're given a problem that requires us to turn a random number that ranges from 1-12 into the appropriate month. When I refer to this problem, it will be marked as the month interpreter problem.

There are many ways in which we can solve this problem. I'll go ahead and share one such solution that we can do using only Lists.

```
22 x = 5
23 #The backslash (\) lets me break up my long line into two lines
24 months = ["January", "February", "March", "April", "May", "June", "July",\
25           "August", "September", "October", "November", "December"]
26 print months[x-1]
```




```
May
```

So here, I went ahead and created a list containing all of my months. I then proceed to print out the appropriate month based on the value of x. Note though that when I pull the element from my list, my index is actually x - 1, not x. This is because our index starts at 0 instead of 1, so we actually have our months lined at 0-11 instead of 1-12. By subtracting 1, I correct the off by one error and am able to easily print out my month.

Let's say we couldn't use this solution though. How would we go about solving this problem? Ideally, we'd like to say:
If x is 1, then print January. If x is 2, then print February… if x is 12, then print December.

If and Else Statements

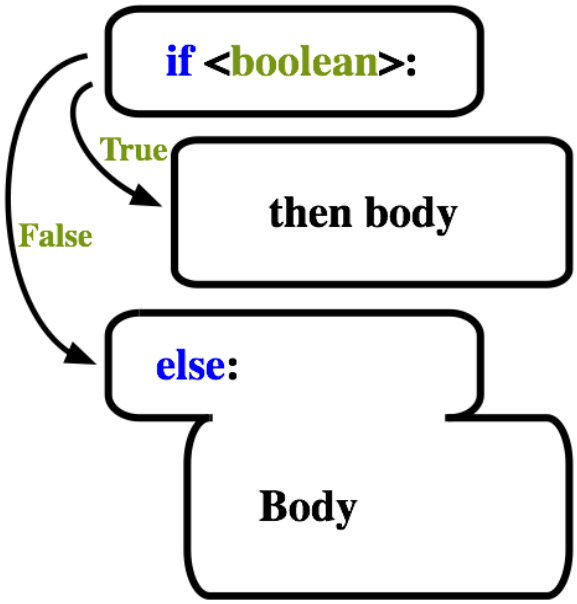
In Python, we can represent this logic with if statements. Let's start things off by first writing some code to check if we should print out January for an arbitrary value of x. See the code below.

```
22 x = 1
23 if x == 1:
24     print "January"
25 else:
26     print "Not January"
```

Reset | Test Run

```
January
```

Here we're printing out "January" and not "Not January" because we set the value of x to be 1. We controlled our output based on a condition. Let's take a close look at the diagram below to get a better idea of how this program flows and operators.



This diagram helps us map out the logic flow of a basic if statement. We have appropriate code written before the if statement. We'll then enter our if condition. The syntax for doing so is:

- if <Boolean>:

where <Boolean> represents the Boolean value that I will insert there. Afterwards, depending on the result of the Boolean value, we'll execute different commands. If the boolean value is True,we'll execute the Then Body. If boolean value is False, then we'll execute the else Body.

Going back to our if statement, we see that we would print "January" if our x is equal to 1, but otherwise we will always print out "Not January" because our Boolean will be False. Let's take this concept and apply it to our problem at hand now. Once we do, our code will look something like this:

```
20
21
22 x = 4
23 if x == 1:
24     print "January"
25 if x == 2:
26     print "February"
27 if x == 3:
28     print "March"
29 if x == 4:
30     print "April"
31 if x == 5:
32     print "May"
33 if x == 6:
34     print "June"
35 if x == 7:
36     print "July"
37 if x == 8:
38     print "August"
39 if x == 9:
40     print "September"
41 if x == 10:
42     print "October"
43 if x == 11:
44     print "November"
45 if x == 12:
46     print "December"
```

Reset | Test Run | Submit | Continue

```
April
```

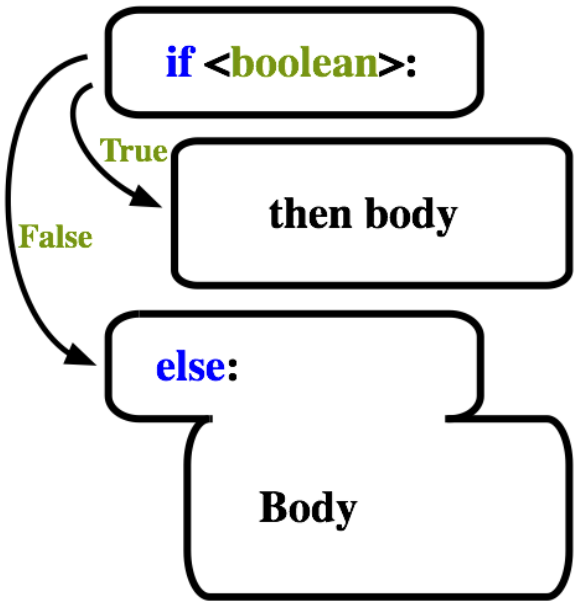
Now we have effective code that will run and print the month based on our variable, x. We'll go through each if statement and then determine the month we should print out, but let's say that we want to provide some feedback. What if x is not a number between 1 and 12? Well, my first instinct was to add an else statement. Let's see what happens when I add the else statement though.

```
x = 4
if x == 1:
    print "January"
if x == 2:
    print "February"
if x == 3:
    print "March"
if x == 4:
    print "April"
if x == 5:
    print "May"
if x == 6:
    print "June"
if x == 7:
    print "July"
if x == 8:
    print "August"
if x == 9:
    print "September"
if x == 10:
    print "October"
if x == 11:
    print "November"
if x == 12:
    print "December"
else:
    print "Number not Valid"
```

Reset | Test Run | Submit | Continue

```
April
Number not Valid
```

Although the value of x was between 1 and 12, our else statement got printed as well. Let's go back to the flow chart (shown on the right) to see why.



As we can see here, our else statement will follow a flow based on the condition given in the if statement directly above it. Because we've created a series of 12 if statements, the else statement doesn't necessarily know which if statement it is based off of. As a result, it will simply default to the if statement directly above it. Since the else statement was applicable only to the if x == 12: statement, it checked if "December" or "Number not Valid" should be printed. Let's look into how we can edit our code so that our else statement would be applicable to all of the if statements and would get printed if and only if the value of x is not between 1 and 12.

The Elif Statement

Let's start off fixing the problem using what we already know. We could fix this by inserting our followup if statements inside of the predecessing if statement. I'll demonstrate this sort of solution with the first three months:

```
22 x = 2
23 if x == 1:
24     print "January"
25 else:
26     if x == 2:
27         print "February"
28     else:
29         if x == 3:
30             print "March"
31         else:
32             print "Not within the first 3 months!"
```

Reset | Test Run | Submit | Continue

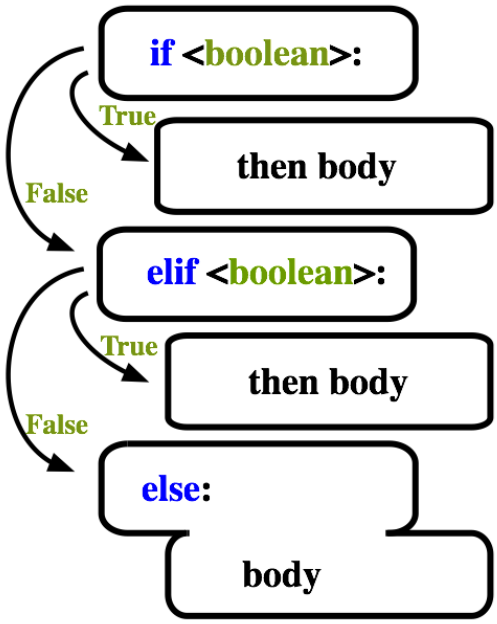
```
February
```

*Note: This is the **WRONG** way to fix our problem!*

By using the code above, our else statement that prints "Not within the first 3 months" is applicable to all the if statements. As you can see here, we were able to print out exactly what we wanted and expected by creating some complicated code with a complicated structure of if statements inside the else statement of our previous if (That's a mouthful, demonstrating that we may be overcomplicating things!). As you might imagine, our program can become really unreadable if we tried to add a lot of if statements in this manner. To get around this, we can write an else if statement all in one line. In Python, we can represent this else if by writing the following:

- elif <Boolean>:

So let's take a look at how our code will now flow before we apply this to our program.



With this diagram, the first thing to note is that we don't need to constantly indent our code every time we want to add another elif. As we see, our code will run down our long if and check each condition until it finally finds a condition which turns out being True. If it doesn't find a condition that's true, then it'll just go ahead and run the else body just like it did before. What's important to keep in mind though, is that as soon as we find a condition that is True, we will not bother going through any of our other elif or else statements.

Below, is an example that will show how when one of our conditions are met, the rest of the conditions we did not go through yet will then be disregarded.

```
x = 2
if x == 1:
    print "I didn't pass!"
elif x == 2:
    print "Made it!"
elif x >= 1: #x is greater than or equal to 1
    print "I wasn't even looked at."
else:
    print "and I wasn't looked at either."
```

Reset | Test Run | Submit | Continue

```
Made it!
```

Here, we printed out "Made it!" just like we'd expect. We did not go through our second elif even though that condition would be True. Let's go ahead and apply this logic to our program now.

```
x = 5
if x == 1:
    print "January"
elif x == 2:
    print "February"
elif x == 3:
    print "March"
elif x == 4:
    print "April"
elif x == 5:
    print "May"
elif x == 6:
    print "June"
elif x == 7:
    print "July"
elif x == 8:
    print "August"
elif x == 9:
    print "September"
elif x == 10:
    print "October"
elif x == 11:
    print "November"
elif x == 12:
    print "December"
else:
    print "This wasn't a month!"
```

Reset | Test Run | Submit | Continue

```
May
```

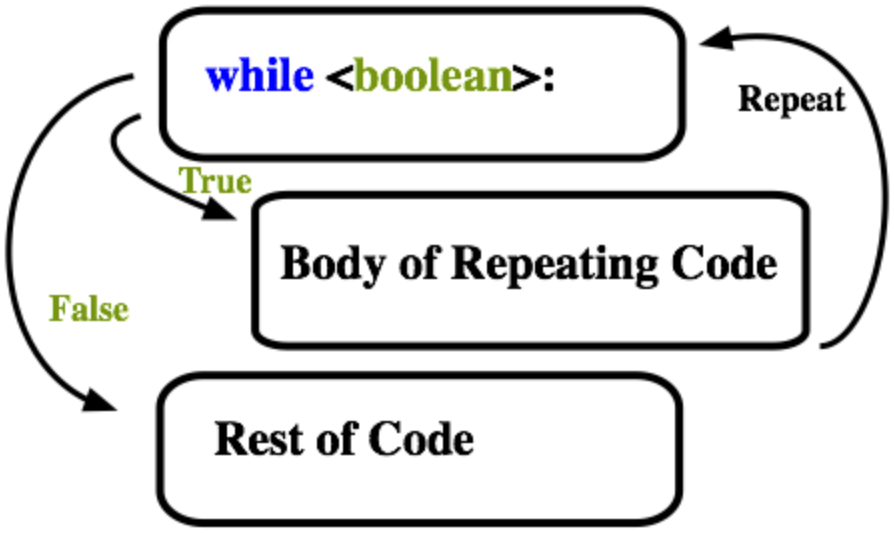
Awesome! Our program now seems to work. We should probably do more thorough testing just to make sure. In the next session, we'll learn to use loops in order to do exactly that.

Loops

When testing our code, we'll want to go through a wide variety of different possible cases. If possible, we want to test each condition that is in our code and then see if there are any edge cases which can cause our code to break. So in this case, we should probably test the numbers from 1 to 12 and then also a couple numbers greater than 12 and numbers less than 1. Let's start off by writing a couple of test values into list form.

- x = [-7, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 55]

Now, while we can reference each individual element in the list manually, it can get really tedious and things could also get very complicated and near impossible when we're dealing with very long lists of indeterminate size. In programming, we can use loops to go over a body of code as many times as we need to. Below is a diagram depicting how our code will flow. If this doesn't make sense, don't worry. We'll go through an example to apply this model.



The first thing we should note is that this code here looks very similar to the if statement. We have our condition, which we then indent in order to convey what will happen should our condition be True. Here's where we start things start to become a little different though. With our loop, we will execute our Body of Repeating Code, but then go back to our condition. If it's True, we'll run through our body again and continue to repeat the cycle until our boolean value is False.

Let's run through an example of when and why we might want to use code. Let's start off by creating a countdown that goes from 5 to 1. Originally, we would probably write our program like this:

```
22 print "5"
23 print "4"
24 print "3"
25 print "2"
26 print "1"
27 print "Blastoff!"
```

Reset | Test Run | Submit | Continue

```
5
4
3
2
1
Blastoff!
```

As we can see, this took up quite a few lines. More importantly though, making changes to this would not be very easy. What if we wanted to change our countdown to countdown from 10 instead? We would probably have to add more print statements that will print the correct number. Let's adjust our code to do a countdown starting from 10 using a loop now.

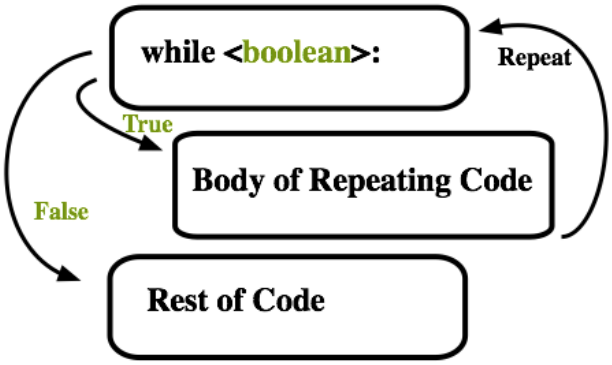
```
22 x = 10
23 while x > 0:
24     print x
25     x = x - 1
26 print "Blastoff!"
```

Reset | Test Run | Submit | Continue

```
10
9
8
7
6
5
4
3
2
1
Blastoff!
```

So with the loop, everything looks a lot more compressed. Let's go through this problem and apply the diagram from before so that we can fully comprehend how it works.

```
21
22 x = 10
23 while x > 0:
24     print x
25     x = x - 1
26 print "Blastoff!"
```



So I started off by defining a variable x, which will keep track of where I am in my countdown. Afterwards, I enter my loop with the condition that I will stay in my loop so long as x > 0. Inside the body, I'lln print x and afterwards decrease x by one. Keep in mind that this will only occur when x is greater than 0 precisely because that's the condition I've set. So I'll go ahead and repeat these steps as x decrements. I'll go through the body a total of 10 times, once when x = 10, when x = 9, x = 8, and onwards up until x = 0. Once I've decremented and x has been decremented all the way to 0, my loop's condition now False. Now that x is not greater than 0, I'll go ahead and execute on the rest of my code, printing out "Blastoff!"

As we can see here, it becomes very easy to make changes to the code now. If I want to do a countdown starting at 1000, I just need to change the starting value of x to be 1000 and the loop will do the rest of the work for me!

Let's go ahead and use our for loop in order to test if our month interpreter works (Note: you'll have to go to the next page in order to see the solution!).

```
my_list = [-7, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 55]
index = 0
while index < len(my_list):
    if my_list[index] == 1:
        print "January"
    elif my_list[index] == 2:
        print "February"
    elif my_list[index] == 3:
        print "March"
    elif my_list[index] == 4:
        print "April"
    elif my_list[index] == 5:
        print "May"
    elif my_list[index] == 6:
        print "June"
    elif my_list[index] == 7:
        print "July"
    elif my_list[index] == 8:
        print "August"
    elif my_list[index] == 9:
        print "September"
    elif my_list[index] == 10:
        print "October"
    elif my_list[index] == 11:
        print "November"
    elif my_list[index] == 12:
        print "December"
    else:
        print "Number not Valid"

    index = index + 1
```

Reset | Test Run | Submit | Continue

```
Number not Valid
Number not Valid
January
February
March
April
May
June
July
August
September
October
November
December
Number not Valid
Number not Valid
```

I've run the code above and I've got a bunch of numbers. On a closer look, it seems like the results seem to overall look correct. Let's take a closer look at how we went through every element of our list so that we could print the output. In this case, there are 3 critical lines of code that we want to take a close look at in order to get a better understanding. I will go ahead and write the code below:

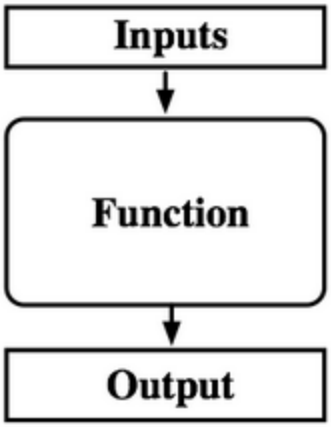
```
1) index = 0
2) while index < len(my_list): #len will fetch the length of the list
3)         index = index + 1
```

1) So here, I go ahead and use index to keep track of the element I want to look at within the list. I'll start with my first element, which is located at the 0th place, hence why I default index to the value 0.
2) The while loop is intended to be used to go through all the elements of my list. In order to achieve that goal, I just constantly check that index is less than the number of elements my_list. Remember, the moment that index == len(my_list), that means that index will no longer reference an element inside the list. For example, if there are 10 elements in the list, then the indices that reference those elements will number from 0-9.
3) In order to make sure that I continue to go through the list, I increment index by 1 to indicate that I'm ready to move on to the next element and start over with my loop.

So as you can see here, the loop is intended to help me test a couple values and see if my code is working properly. Luckily, the code seems to be working properly, but how can we have other programmers and users effectively use our code? Well, ideally, we want them to dictate to us what the values that are passed in to our month interpreter will be. Let's go ahead and take a look at the next section to see how we can use functions to effectively do that.

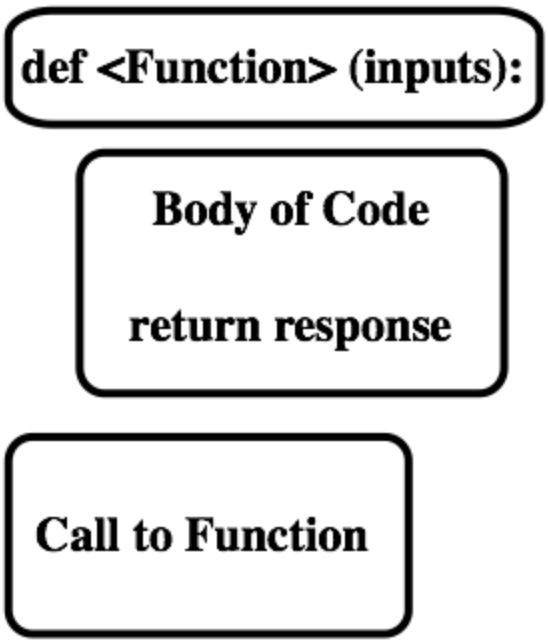
Functions

In the section on if statements, we wrote a fully functional program that told us the month based on a number input. We then took advantage of loops in the following section to write tests for the program so that we can make sure that it's working as intended. Now we have one glaring problem that we want to deal with: how can we have other people take full advantage of this program? Ideally we want them to simply tell the program a number. Our program will then interpret the number and spit out the corresponding month to that number. Functions allow us to do exactly that.



A function can be likened to a magical generator. The generator will need to be fed in some components, and based off those components, it'll go ahead and do work. Once all that work is done, the generator will give me back a resulting product: the output. So the solution that we want to act on is to create a product that will function as shown in the diagram to the right. I'll go ahead and build my month interpreter function so that I can have other programmers and users use this function by referencing it and passing in a number as input. Based on that number, I'll give back the name of the month.

Let's take a look at the diagram below to see how functions work. So as we can see here, we go ahead and indicate that we're writing a function through the keyword, def. Afterwards, go ahead and give your function a unique name of your own choosing and specify how many inputs you will have by creating variables for each input. Here is a sample function definition below:

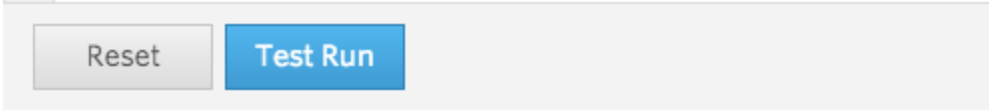



```
30 def my_own_function(a, b, c):
31     return "This is my function that takes 3 inputs!"
```

Now that I've created my function, I can go ahead and reference it by simply stating the name and passing in inputs. Once that happens, the function will go ahead and run the code inside. Using the keyword, return, it'll then pass my intended output over to what called the code so that it can

be used. Here is an example of how I would call the function above and afterwards print out the results:

```
38 def my_own_function(a, b, c):
39     return "This is my function that takes 3 inputs!"
40
41 x = my_own_function(1, 2, 3)
42 print x
```




```
This is my function that takes 3 inputs!
```

As we can see here, I went ahead and called the function by calling its name. I passed in three inputs like the function asks for (those values are stored into variables named **a**, **b**, and **c**). The function then did work and fed me an output, which I stored in my variable, x, which I then proceeded to print out so that we could see the results of our code.

Let's go ahead and take our month interpreter solution and place it inside a function, which we'll call month_interpreter (Note: you'll have to go to the next page in order to see the solution!)

```
def month_interpreter(x):
    if x == 1:
        return "January"
    elif x == 2:
        return "February"
    elif x == 3:
        return "March"
    elif x == 4:
        return "April"
    elif x == 5:
        return "May"
    elif x == 6:
        return "June"
    elif x == 7:
        return "July"
    elif x == 8:
        return "August"
    elif x == 9:
        return "September"
    elif x == 10:
        return "October"
    elif x == 11:
        return "November"
    elif x == 12:
        return "December"
    else:
        return "Number not Valid"


print month_interpreter(5)
```

Reset | Test Run | Submit | Continue

```
May
```

I went ahead and copied the original function we wrote without test cases into the body of a function. Finally, I made sure to replace all of my print statements with return statements so that it will properly provide an output that can be used by the caller of the function.

Let's talk about the benefits to using a function here now. Whereas we had to previously write the block of code within where we want to use it, we can now simply reference the name of the function to use it. This offers a lot more flexibility for us as we write code others can use. Now, instead of having to understand the code inside, the user simply needs to understand the intentions of the function in order to effectively use it. In essence, we have effectively abstracted our code away, so that people only need to worry about the big picture.

Conclusion

Praise the sun! We have effectively condensed what is often taught over the span of a couple weeks into about 23 pages of content, so great job, keep at it and I hope you took things away from here! You should now be familiar with the basics of programming. These concepts will have helped prepare you for the Introduction to Object Oriented Programming course, which will delve into the concept of abstraction, which we briefly touched upon in the Functions section. If you would like to learn more about various concepts in Computer Science though, please feel free to check out the Introduction to Computer Science course as well!

Good luck and stay Udacious!