Name: _____

Week 4:
Functions, Strings, and Arrays

Functions

Sometimes, when we're writing code, we want to be able to use the same code in more than one place. For example, several times so far we've wanted to ask the user for their name and print it. Each time, we write code like

```
string_name = input("What's your name?")
print("Hello " + string_name + "!")
```

Wouldn't it be nice if, rather than having to rewrite this code each time we wanted to ask the user for their name and greet them, we could just write it once and then tell the computer to use that same code multiple times? Well, the good news is, we can! This is what is called a **function**. A function is a way of telling Python that you want to run code that is located somewhere else in the file. (Or, as we'll learn later, even from other files!) To make a function, we write the word `def`, then a space, then the name of the function, and then a pair of parentheses and a colon "():". For example, to define a function named "foo", we would write

```
def foo():
```

and to define a function called "bar", we would write

```
def bar():
```

After the definition line we put the code we want to be able to reuse, followed by the word `return` on its own line, all indented by four spaces. (As usual, this is additive, so any statements in an `if` or `while` will be indented by four spaces for the function, plus the four spaces for the `if` or `while`.) As an example, if we wanted a function named "greeting" that would allow us to reuse our greeting code from above, we would write this

```
def greeting():
    string_name = input("What's your name?")
    print("Hello " + string_name + "!")
    return
```

Note that when you `def`ine a function, Python will not execute those lines! It just remembers where they are, so that it can execute them if you call them later. But if you don't call your function, these lines will never be run. Fortunately, calling a function is very simple - we just

write the name of the function followed by a pair of parentheses. So to call our greeting function, we would write

```
greeting()
```

And that's it! When Python sees a word followed by "()", it knows it's a function. So it goes and looks for where we've written the `def` for that function, and jumps to the first line in the `def`inition. It will execute the lines in that function until it reaches the `return` line, at which point it will `return` back to the line that the function was called from. Let's see this in action:

```
def greeting():
    string_name = input("What's your name?")
    print("Hello " + string_name + "!")
    return

print("Starting program.")
greeting()
print("Ending program")
```

Line being executed:	Explanation:
`def greeting():`	This tells Python we're defining a function named `greeting`. It remembers where this code is for when we call `greeting`, and then skips all of the indented code. Remember that Python does not run the lines inside function definitions until they are called!
`print("Starting program.")`	Python prints "Starting program."
`greeting()`	Python sees the "()" after the word "greeting", and knows we want to execute a function named "greeting". It remembers the location of "greeting" from when it saw the `def` statement earlier, so it jumps to there.
`def greeting():`	For now, the definition line does nothing when we call the function, but later on we'll see that's not always the case. After finding the definition, Python moves on to the first indented line in the function.
`    string_name = input("What's your`	Prints out "What's your name?" to the user, and waits for them to type in a response, then stores that response (as a string, because

`name?")`	`input` is always a string!) into `string_name`.
`    print("Hello " + string_name + "!")`	Prints out a greeting to the user.
`    return`	Tells Python that our function is done, and that it should go back to where it was before the function was called.
`greeting()`	Python returns to the line where the function was called. Since the call is done, it moves on to the next line.
`print("Ending program.")`	Prints out “Ending program.”

So the end result is the same as if we’d written

```
print("Starting program.")
string_name = input("What's your name?")
print("Hello " + string_name + "!")
print("Ending program")
```

This will usually be the case with functions - you can imagine that the lines of code inside the function are just replacing the line where you call the function from.

One last note on functions: you can’t use variables inside your function that you’ve defined outside your function. You also can’t use variable names outside your function that you’ve defined inside your function. So, for instance writing this would result in an error:

```
bar = "Hello, world!"

def foo():
    print(bar)
    return

foo()
```

But this would be fine:

```
def foo():
    bar = "Hello, world!"
    print(bar)
    return

foo()
```

Variables in Python are restricted to the function that they appear in, or to being in no function. This fact means that you can reuse variable names, as long as you don't use the same name twice inside any one function. So this code

```
def foo():
    bar = "a"
    print(bar)
    return

bar = "b"
foo()
```

will print "b", and not throw an error, even though we've defined `bar` twice.

Example Problems:

`def say_hi():` `  print("Hello!")` `string_name = "Howard"` `say_hi()`	What is printed?
`def ask_name():` `  string_name = "Howard"` `ask_name()` `print("Hello " + string_name)`	What is printed?
`def print_letters():` `  print("a")` `  print("b")`	What is printed?

```
    print("c")

int_i = 0
while int_i < 3:
    print_letters()
    int_i += 1
```

Value Returning Functions

Functions are very useful for avoiding writing the same code over and over, but sometimes we want to do something with the value that our repeated code calculates. For example, we've often wondered if the value the user returned is equal to some other value, and used code like

```
input("What's the password?") == "password"
```

as the condition on `if` statements or `while` loops. Could we put that in a function, so that we could give it a nice name like "guess_is_correct" and avoid having to write it all the time? Well, we could try writing something like this:

```
def guess_is_correct():
    input("What's the password?") == "password"
    return
```

But that won't quite work. The problem is that while this will ask the user for input, and check that input against the desired value, it won't do anything with it. We compare the input to "password", but then we just `return` back to the line that called `guess_is_correct()` without doing anything with that result! The way we handle this problem is to `return` a value. Instead of writing the word `return` all by itself, we write `return` and then a space, and then a value. When Python calls the function, if there's a value after the `return` statement, Python will replace the function call with the value that is `return`ed. So, we write this instead:

```
def bool_guess_is_correct():
    return input("What's the password?") == "password"

bool_correct = bool_guess_is_correct()
if bool_correct:
    print("Correct!")
else:
    print("Incorrect!")
```

(Note that if we are `return`ing a value from a function, we prefix the function name with the value type we will be `return`ing, just like how we prefix our variable names with the value type.)

When Python gets to the

```
bool_correct = bool_guess_is_correct()
```

line, it will create a new variable called `bool_correct`, and then attempt to store into it whatever is on the right hand side of the equals sign. When it looks at the right hand side though, it sees the parentheses and knows that means it's going to need to evaluate a function. So it jumps to the function `def`inition, and then runs the line

```
    return input("What's the password?") == "password"
```

But now, because there is a value after the `return`, instead of immediately jumping back, it will calculate the value after the `return`, asking the user for input and performing the "==" operation. Once it has the resulting boolean value, it will take that value and effectively *replace* `guess_is_correct()` with that value, resulting in the calling line looking, to Python, something like

```
bool_correct = True
```

if the value after `return` evaluated to `True`.

Since Python will run functions and replace them with the returned value anywhere it sees a function call, we can even make our code shorter, and just say

```
def bool_guess_is_correct():
    return input("What's the password?") == "password"

if bool_guess_is_correct():
    print("Correct!")
else:
    print("Incorrect!")
```

leaving out the `bool_correct` variable entirely.

Example Problems:

`` def mult_two(): int_a = 5 int_b = 7 return int_a * int_b int_result = mult_two() print("The result is " + str(int_result)) ```	What is printed?

`` def repeat_word(): str_word = "cows" int_times = 3 return str_word * int_times str_repeated_word = repeat_word() print("The repeated word is " + str_repeated_word) ```	What is printed?
`` def is_valid(): int_number = 3 return int_number < 10 if is_valid(): print("Valid!") else: print("Invalid!") ```	

Function Parameters

Returning values is useful, but sometimes we're going to want to pass values into our functions, rather than asking the user for them. For instance, if we were in a `while` loop counting from 0 to 20, and wanted to print out only the even numbers, we might write something like this:

```
int_i = 0
while int_i < 20:
    if int_i % 2 == 0:
        print(str(int_i))
    int_i += 1
```

We've seen that `int_i % 2 == 0` before though! We end up writing that condition a lot it seems. Let's make it into a function instead, so we can reuse it!

```
def bool_is_even():
    return int_value % 2 == 0

int_i = 0
while int_i < 20:
    if bool_is_even():
        print(str(int_i))
```

```
    int_i += 1
```

Wait a minute though, that's not going to work! We never say create a value named `int_value`, and we know we can't use variables inside of a function that are declared outside of it, so even renaming it to `int_i` wouldn't help. How can we get the value we want to check for evenness inside of our function?

To do this, we use what are called **function parameters**. This is basically a fancy way of saying that when we write a function, we can make some special variables and say that the person calling our function has to tell us what they should be when they call the function. To do this, we put the names of the variables that we want the caller to set in our parentheses like this:

```
def bool_is_even(int_value):
    return int_value % 2 == 0
```

Then, when the caller is calling the function, they put the value they want to set that variable to inside the parentheses of the call:

```
int_i = 0
while int_i < 20:
    if bool_is_even(int_i):
        print(str(int_i))
    int_i += 1
```

We don't have to stop at one variable, either! We can use multiple:

```
def int_multiply(int_a, int_b):
    return int_a * int_b

int_i = 1
int_j = 1
while int_i <= 5:
    while int_j <= 5:
        print(str(int_multiply(int_i, int_j)))
        int_j += 1
    int_i += 1
```

This function above will print out all of the possible multiplications of numbers between 1 and 5 inclusive. When we see `int_multiply(int_i, int_j)`, what that's telling Python is that the values of `int_a` and `int_b` should be set to the same values currently in `int_i` and `int_j`, respectively.

As a last note about function parameters, make sure that you always have the same number of values in the function `def`inition as you do in the function call! You'll get an error if you do something like put three values in the `def`inition and only two in the call, or vice-versa.

Example Problems:

`` def mult_two(int_a, int_b): return int_a * int_b int_c = 5 int_d = 7 int_result = mult_two(int_c, int_d) print("The result is " + str(int_result)) ```	What is printed?
`` mult_three(int_a, int_b, int_c) return int_a * int_b * int_c int_c = 5 int_d = 7 int_result = mult_two(int_c, int_d) print("The result is " + str(int_result)) ```	What is printed?
`` def check_number(int_num): if int_num % 2 == 1: return "good" else: return "bad" def is_good(str_value): if str_value == "good": return True else: return False int_secret = 7 if is_good(check_number(int_secret)): print("All good") else: print("Not so good.") ```	What is printed?

Strings

We've learned some about strings already, and used them a fair amount, but now that we know about functions, it turns out there are a lot more cool things we can do with strings.

Built-In Functions

Python has a number of built-in functions for strings that make our lives easier. They are listed below, as well as how to use them. Note that there are two different ways we call these functions, some functions taking the string as an argument, like we learned about above, and some functions are called by typing a period after the string, then calling the function with no arguments. We'll learn the reason for this later on, when we get to **classes**. For now, the simplified explanation of why they differ is that the functions that take the string as an argument can often be called on things that are not strings as well, while the ones that use the .functionName() format are string-specific.

Function:	Explanation:
len("abc")	Returns the length of the string. In this example, it would return, or be replaced by, 3.
"abc".upper()	Converts the string to uppercase. In this case, it would return "ABC".
"ABC".lower()	Converts the string to lowercase. In this case, it would return "abc".
"a" in "abc"	A new operator that tells us if the first string is contained in the second one. Returns a boolean telling us whether or not the first string is a substring of the second. In this case, it would return True. We often use this as the condition on `if` or `while` statements.
ord('a')	ord() takes a single character as its input, and returns the the integer value of that character. In this case, it would return 97.
chr(97)	Converts an integer into the corresponding character. In this case, it would return 'a'.

Now, you'll notice that the last two functions are about characters, even though we're talking about strings. Why is that? Well, if you remember, strings are really just collections of characters. So the string "Hello", to Python, would look like this:

'H'	'e'	'l'	'l'	'o'

```
string_hi = "hello"
print(str(string_hi[0])) # Prints "h"
```

When we see it like this, it's natural to ask if we can get individual characters out of the string. The answer is yes! We just use an integer in square brackets after the string to tell Python which character we want, and it will fetch it for us, like "abc"[2]. The main catch here is that Python starts counting from 0, so to get the 'H' character from "Hello", we would write "Hello"[0], and to get the 'o' we would write "Hello"[4]. This can be done with literal strings (strings with double quotes around them) or variables that contain strings! We can also use integer variables for the number in the brackets. So this is valid:

```
string_name = "Howard"
int_position = 0
print(str(string_name[int_position]))
```

How is this useful? Well, many of you might be familiar with the Caesar Cipher, but for those who haven't seen it before, the Caesar Cipher is a simple way of making secret messages - all you do is take each letter of the alphabet, and replace it with the next one. So 'a' becomes 'b', 'b' becomes 'c', and so on. Then, when your friend has received your secret message, they just reverse the process, turning 'b' into 'a' and so on, and they have the original message back!

Using the fact that we can get the individual characters from a string in Python, we can make a Caesar Cipher! There is one other thing we need to know in order to make it though. When we convert a character to an integer using ord(), the characters are in alphabetical order. This fact means that if we increment the ord() of a character and then convert it back, we'll have the next character in the alphabet!

What we want to do is to have a loop that runs a number of times equal to the number of characters in the string, and get the next character each time we go through the loop. We can use a counter for that! Once we have a character, we want to convert it to an integer so we can add one to it, and then convert it back to a character and put it into a new string, our encoded string. In Python, that would look something like this:

```
string_to_encode = input()
int_i = 0
string_after_encoding = ""
while int_i < len(string_to_encode):
```

```
    char_to_increment = string_to_encode[int_i]
    int_to_increment = ord(char_to_increment)
    int_to_increment += 1
    char_after_increment = chr(int_to_increment)
    string_after_encoding += str(char_after_increment)
    int_i += 1
print(string_after_encoding)
```

Some things to note about this code: We build up our result by repeatedly appending characters to string_after_encoding - this is a pattern you will use frequently. We use a "counter" variable, int_i, that starts at 0 and goes until `len(string_to_encode) - 1`, adding one to it every time we go through the loop, and then using that counter to get the character from string_to_encode in order to make sure we process every character.

These string tools will enable us to do a lot more cool things with strings!

Example Problems:

`str_hi = "hello"` `print(str(str_hi[3]))`	What is printed?
`chr_a = 'a'` `int_a = ord(chr_a)` `chr_not_a = chr(int_a + 5)` `print(str(chr_not_a))`	What is printed?
`str_dog = "dog"` `int_i = 0` `while int_i < len(str_dog):` `  print(str(str_dog.upper()[int_i]))` `  int_i += 1`	What is printed?

Arrays

Seeing how useful it is to be able to keep multiple characters in a string, you might be wondering if there's any way that we can keep a list of other types of data. The good news is that there is! They are called **arrays**, though Python refers to them as lists. (There is technically a difference between arrays and lists, but you'll learn that later on and don't need to worry about it right now.)

Arrays work very similarly to strings! We use square brackets to denote items in an array, separating the items by commas. So we would write [1, 2, 3, 4, 5] to denote the array containing the integers 1, 2, 3, 4, and 5. To Python, the array would look like this:

1	2	3	4	5

And we can get the integers out in the same way, by writing the location of the item we want in square brackets after the array: [1, 2, 3, 4, 5][0]. Obviously this is somewhat silly, since we could have just written 1, but it gets more useful when we store our arrays into variables, like this:

```
floats_numbers = [5.0, 0.0, 0.0]
float_last = floats_numbers[2]
print(str(float_last)) # Prints 0.0
```

You'll notice that we can have more than one of the same value in an array, just like we can have more than one of the same character in a string! You can also store any value type in an array. Even strings! ["abc", "def", "ghi"] is totally valid. (You can even put arrays in your arrays! But don't worry about that yet.)

Just like we write the empty string as "", we write the empty array as []. For working with arrays, we have a few special functions:

len(["some", "strings", "here"]) # Returns 3	len() tells us the length of arrays, too! In this case, it would return 3. Note that what it returns is the number of items in the array - the fact that the items are strings with their own lengths doesn't matter, because there are still only 3 of them.
[1, 2].append(1) # Results in [1, 2, 1]	.append() will append an item to the array, or put it at the end of the array. In this case, our array would become [1, 2, 1]. Note that .append() does **not** return the value, so unless we use it on an array that's stored in a variable, the computer will calculate the

	appended array, but immediately forget it.
[“a”, “c”].insert(1, “b”) # Results in [“a”, “b”, “c”]	.insert() takes two arguments, an integer representing where to insert, and the value to insert. It will put the value provided into the slot specified (remembering that we start counting from 0!). In this case, we would end up with [“a”, “b”, “c”]. Just like .append() though, this function does not return its value, so since we don’t have it in a value, Python will simply calculate it and immediately forget it.
[5.3, -4.4, 1.0, 0.2].pop() [5.3, -4.4, 1.0, 0.2].pop(0) # First one results in [5.3, -4.4, 1.0] # Second one results in [-4.4, 1.0, 0.2]	.pop() can either take no arguments, or one integer. If no arguments are provided, it removes the last item in the array. If an integer is provided, it removes the item at that position. As with the previous functions, this change will be lost if not used on an array stored in a variable.
7 in [1, 2, 3] # Returns False	Just like we can use the `in` operator to see if a character is in a string, we can use it to see if a value is in an array. In this case, it would return False, because 7 is not in the array.

One thing we can do with arrays that we can’t do with strings is to *change* items in the array. If we wrote

```
string_word = "dog"
string_word[0] = "b"
```

we would get an error. However, if we do this with an array, like so:

```
floats_numbers = [1.1, 2.2, 3.3]
floats_numbers[0] = 0.0
```

it will succeed, and floats_numbers will become [0.0, 2.2, 3.3].

In the list of functions used for arrays above, I mentioned that most of them require the array to be stored in a variable to work. This is because

```
ints_array = ["a", "c"].insert(1, "b")
```

won’t work. It won’t throw an error, but when Python evaluates the .insert() function on the right hand side, it won’t return anything. That means we won’t have a value to put in our variable, so we will just get an empty variable, or a box with nothing inside it. What we should do instead is something like this

```
ints_array = ["a", "c"]
ints_array.insert(1, "b")
```

now ints_array will be [“a”, “b”, “c”] like we want it to be. A good rule of thumb is that you should only use literal arrays (arrays where you write out the brackets and the values inside the array) when you’re initially creating your array variables. Other than the creation, you should always reference your arrays by using a variable.

We can convert arrays to strings with str(), just like with our other value types. The string we get will look something like “[1, 2, 3]”.

One last thing - for our value type prefix, when using arrays, we make the type plural. So int becomes ints, str becomes strs, and so on.

Example Problems:

`` ints_numbers = [5, 3, 1] print(str(ints_numbers[1])) ```	What is printed?
`` ints_numbers = [0, 0, 7] int_i = 0 while int_i < len(ints_numbers): ints_numbers[int_i] += 1 int_i += 1 print(str(ints_numbers)) ```	What is printed?
`` strs_animals = ["puma", "cat", "lion"] str_pet = "cat" int_i = 0 while int_i < len(strs_animals): if strs_animals[i] == str_pet: print(strs_animals[i]) ```	What is printed?
`` strs_animals = ["puma", "cat", "lion"] str_pet = "cat" ```	What is printed?

```
int_i = 0
while int_i < len(strs_animals):
    if strs_animals[i] == str_pet:
        print(strs_animals[i])
    int_i += 1
```

```
def get_2nd(any_array):
    return any_array[1]

print(str(get_2nd([5.4, 1.2, 4.3])))
```

What is printed?