Python Battleships Review

Chami Lamelas

August, 2021

In this document, I will review the Python syntax you will need to complete the battleships assignment.

1 Variables and Primitive Data Types

In this section, I will talk about primitive data types, viewing data types in Python, and casting.

1.1 Declaring Variables

You can name a variable anything other than a Python reserved word. For example, this code creates an integer variable `int_var` with value 5, a floating point variable `float_var` with value 3.14, a string variable with value "abc", and a boolean variable with value `True`.

```
int_var = 5
float_var = 3.14
string_var = "abc"
bool_var = True
```

Boolean variables can take on two values `True` or `False`.

1.2 Viewing Data Types

If you ever want to see the type of a variable, use the function "type()" which will return the type of the variable as a string. For example, we can print the types of the variables defined above. The `print` function will be discussed in more detail later.

```
print(type(int_var))
print(type(float_var))
print(type(string_var))
print(type(bool_var))
```

This outputs

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

1.3 Casting

Strings are enclosed in quotation marks but can themselves contain numbers. To convert a string containing a number to a number, you need to cast it. In the code below, we have a string variable `str10` which holds the string "10". We then cast it to an integer to get the resulting integer variable `int10`.

```
str10 = "10"
int10 = int(str10)
```

We can confirm that they indeed have different types by printing their types.

```
print(type(str10))
print(type(int10))
```

This outputs

```
<class 'str'>
<class 'int'>
```

For now, the `int()` function may seem useless, but all user input in Python is read as strings. Therefore, if you ever want to use user input as integers, you will have to cast them.

You can also cast from integers back to strings using the `str()` function. This code casts the integer variable `int20` to string variable `str20`.

```
int20 = 20
str20 = str(int20)
```

We can again print their types

```
print(type(int20))
print(type(str20))
```

This outputs

```
<class 'int'>
<class 'str'>
```

`str()` will be useful when we want to make integers as part of strings. This will be discussed in more detail in the string concatenation and printing sections below.

1.4 Constants

In other languages like Java and C, you can specify that some variables are constant and will never be changed after they are initialized. Python does not provide this capability, but you can signify variables as constants by making them entirely uppercase.

```
INCHES_IN_FOOT = 12
PI_APPROX = 3.14
```

You are still able to change these variables later on, but uppercase is meant to signify to yourself or other programmers, that they should not be changed. Non-constant variables should be lowercase.

2 Integer and String Operations

In this section, I will discuss some simple operations that can be performed on integers and strings.

2.1 Integer Addition and Subtraction

You can add two integers using (+) and subtract them using (-). In this code, we store the result of adding 10 and 5 in variable `a` and then print the result.

```
a = 10 + 5
print(a)
```

This outputs

```
15
```

We can also add two integer variables. This code creates two integer variables `b` and `c`, adds them together into variable `d`, then prints the result.

```
b = 6
c = -1
d = b + c
print(d)
```

This outputs

```
5
```

You can subtract integers in the same way, as in the following code.

```
f = 3 - 9
print(f)


g = 4
h = 2
i = g - h
print(i)
```

This outputs

```
-6
2
```

We can multiply integers in the same manner. However, division works slightly differently. Consider the following code

```
j = 6
k = 3
l = j/k
print(l)
```

This outputs

```
2.0
```

That is, `l` is a floating point variable. Similarly, if we perform the division

```
m = 2
n = 3
p = m/n
print(p)
```

`p` is $0.\overline{6}$.

2.2 Updating Shortcut

It is very common that you will be updating a variable to be n more or n less than its current value. For example, suppose you have a variable `x` that is 10. Later on, suppose you want to update the variable `x` to be 5 more than it's current value. You can write

```
x = x + 5
```

which properly updates `x` to be 15. However, Python provides a shortcut

```
x += 5
```

Python provides similar shortcuts for subtraction, multiplication, and division. Instead of writing

```
y = y - 5
y = y * 5
y = y / 5
```

we can write

```
y -= 5
y *= 5
y /= 5
```

2.3 String Concatenation

If you want to combine strings one after another, you can concatenate them using the + operator. Suppose we have the following string variables

```
first_name = "John"
last_name = "Smith"
```

Now, we want to combine them into a new string variable `full_name` that will be made up of the first name followed by the last name separated by a space. We can create this variable as follows

```
full_name = first_name + " " + last_name
```

If we print this, it will display

```
John Smith
```

It is important to note that we do not have to declare a space " " as a variable. Whenever a string appears in code surrounded by quotation marks, it is called a string literal. We can concatenate string variables and literals in any order.

We cannot concatenate a string with an integer. For example, suppose we again had the string variables

```
first_name = "John"
last_name = "Smith"
```

But we also had John Smith's age

```
age = 45
```

Now, suppose we want to create a string variable that has the first name followed by the last name, followed by the age, each separated by a space. First, try concatenating them as is with +:

```
full_name_and_age = first_name + " " + last_name + " " + age
```

This will generate an error

```
TypeError: can only concatenate str (not "int") to str
```

`str` is the Python string data type and `int` is the integer data type. This error is telling us that we have a variable with the wrong type in the concatenation. To fix this issue, we need to cast age to a string before concatenating it.

```
full_name_and_age = first_name + " " + last_name + " " + str(age)
```

This code will put the first name, followed by a space, then the last name, then another space, then the string representation of the age into `full_name_and_age`. Note that we do not need to store `str(age)` in a variable, we can do it in the declaration of `full_name_and_age`.

For another example, suppose we have three integers storing the year, month, and date of someone's birthday.

```
bday_year = 2020
bday_month = 12
bday_day = 11
```

Suppose we want to turn them into a MM/dd/yyyy string. That is, the month followed by a /, then the day, another /, and then the year (with no spaces!). We can do this with the following code

```
bday_str1 = str(bday_month) + "/" + str(bday_day) + "/" + str(bday_year)
```

Note that we have to cast each of the integer variables to strings before putting slashes between them.

We could also easily create a yyyy-MM-dd date string from these same three variables

```
bday_str2 = str(bday_year) + "-" + str(bday_month) + "-" + str(bday_day)
```

3 Control Flow I: Conditional Statements

In thie section, I will discuss boolean conditions and how they are used to build if-elif-else statements.

3.1 Comparison Operators

Integers can be compared using a set of operators

- `<` : less than
- `>` : greater than
- `<=` : less than or equal to
- `>=` : greater than or equal to
- `==` : equal to (this applies to both strings and integers)
- `!=` : not equal to

Note the difference between the equality operator == and the assignment operator =.

3.2 Primitive Boolean Conditions

A boolean expression is any expression that evaluates to `True` or `False`. We can use comparison operators to build boolean expressions. For example,

```
b1 = 2 < 5
b2 = 2 > 5
b3 = 3 >= 1
b4 = 3 <= 1
b5 = 2 == 1
b6 = 2 != 1
```

`b1` to `b6` are all boolean variables. Printing them outputs

```
True
False
True
False
False
True
```

3.3 Building Compound Boolean Conditions

You can build more complex boolean conditions using `and` and `or`. Suppose that `b1` and `b2` are boolean expressions.

The compound expression `b1 and b2` is evaluated in this manner

1. Check if `b1` is `True`. If so, go to step 2. Otherwise, `b1 and b2` is `False`.
2. Check if `b2` is `True`. If so, `b1 and b2` is `True`. Otherwise, `b1 and b2` is `False`.

The compound expression `b1 or b2` is evaluated in this manner

1. Check if `b1` is `True`. If so, `b1 and b2` is `True`. Otherwise, go to step 2.
2. Check if `b2` is `True`. If so, `b1 and b2` is `True`. Otherwise, `b1 and b2` is `False`.

When building more complex compound expressions with nested `and`s and `or`s, make sure to enclose expressions in parenthesis to ensure expressions are evaluated in the way you want.

For example, suppose someone is allowed to attend an event if they are at least 18 or if they are at least 16 with an adult. For simplicity, suppose we have already determined the age and companion in an integer variable `age` and boolean variable `with_adult` respectively. We can construct a boolean variable `allowed` that stores whether the person is allowed into the event

```
allowed = (age >= 18) or (age >= 16 and with_adult)
```

Note here that we collect the two separate "allowed" conditions in parenthesis. `(age >= 18)` means the person is an adult and `(age >= 16 and with_adult)` means the person is at least 16 and accompanied by an adult. One could ask, why don't we need to write

```
allowed = (age >= 18) or (age < 18 and age >= 16 and with_adult)
```

However, as noted in the evaluation order for `or` listed above, the second expression will only be evaluated if the first expression is false. In this case, if we get to evaluating `age < 18 and age >= 16 and with_adult`, that means `(age >= 18)` evaluated to `False` already. Therefore, `(age < 18)` is redundant and the second expression can be simplified to `(age >= 16 and with_adult)`.

3.4 Negating Boolean Conditions

In the section on comparison operators, we introduced the not equals operator `!=`. However, we can also negate any boolean expression (or variable) with the keyword `not`. For example, we can create the boolean variable `rejected` as the opposite of `allowed`.

```
rejected = not allowed
```

Or, we can write it as the negation of the boolean expression used to define `allowed`

```
rejected = not ((age >= 18) or (age >= 16 and with_adult))
```

It is important to note here that we want to negate the entire boolean expression `(age >= 18) or (age >= 16 and with_adult)` so we enclose it in parenthesis before applying `not` to it.

3.5 if-elif-else Statements

We can use boolean expressions to build `if-elif-else` statements. A simple `if` statement can be constructed

```
if allowed:
    print("Welcome to the event!")
```

This code will print "Welcome to the event!" whenever the boolean variable `allowed` is `True`. We do not need to write `allowed == True` because `allowed` on its own evaluates to a boolean value and can be verified by an `if` statement. This is known as *boolean zen*.

The print statement is known as the body of the if statement. All of the lines in the body of the if statement must be indented by 1 from the if statement. The body of the if statement is executed when the boolean condition is `True`.

The boolean condition check by the if statement does not need to be stored in a variable. For example, the above if statement is identical to

```
if (age >= 18) or (age >= 16 and with_adult):
    print("Welcome to the event!")
```

Now, suppose that we want to print "Sorry, you cannot enter!" in the event if someone does not match the criteria to enter. We can implement this with an `else` statement.

```
if (age >= 18) or (age >= 16 and with_adult):
    print("Welcome to the event!")
else:
    print("Sorry, you cannot enter!")
```

Like with an if statement, the body of the else statement must be indented by 1 tab from the else.

Now, suppose that we want to print "Welcome to the event!" if someone is an adult, "Welcome to the event, stay with your adult!", if someone is at least 16 with an adult, and "Sorry, you cannot enter!" otherwise. We can implement this with an `if-elif-else` statement.

```
if age >= 18:
    print("Welcome to the event!")
elif age >= 16 and with_adult:
    print("Welcome to the event, stay with your adult!")
else:
    print("Sorry, you cannot enter!")
```

Note that we have broken up the compound boolean condition used to define `allowed` because we require two different print statements. An `elif` statement is only checked if all preceding `if` and `elif` statements have evaluated to `False`. The body of the `elif` statement, like `if` and `else` statements must be indented by 1 tab. The body will only be run if the `elif` condition is `True`. The `else` body is executed if all preceding `if` and `elif` statements were checked and evaluated to `False`.

Here, a more complicated `if-elif-else` statement is used to print the appropriate education based on age.

```
if age < 5:
    print("Pre-school")
elif age < 13:
    print("Middle school")
elif age < 18:
    print("High school")
elif age <= 22:
    print("College")
else:
    print("Post-college")
```

Suppose we had instead implemented this with

```
if age < 5:
    print("Pre-school")
elif age >= 5 and age < 13:
    print("Middle school")
elif age >= 13 and age < 18:
    print("High school")
elif age >= 18 and age <= 22:
    print("College")
else:
    print("Post-college")
```

This is equivalent, but redundant. In order to reach `age >= 5 and age < 13`, `age < 5` needed to have evaluated to `False`. Or in other words, `age >= 5` would always evaluate to `True` if `age < 5` was passed. The conditions `age >= 13` and `age >= 18` appear in the third and fourth `elif` statements are redundant for the same reason.

Now, suppose we had implemented this with

```
if age < 5:
    print("Pre-school")
if age < 13:
    print("Middle school")
if age < 18:
    print("High school")
if age <= 22:
    print("College")
else:
    print("Post-college")
```

Consider the case where `age = 12`. This would print

```
Middle school
High school
College
```

This is because a list of `if` statements do not work the same as a list of `elif` statements. Each `if` statement will be checked regardless of whether `if` statements are evaluated to `True`. In the case where `age = 12`, `age < 13`, `age < 18`, and `age <= 22` all evaluate to `True`. Hence, the three corresponding education levels are printed.

3.6 Ternary Operator (if-else shortcut)

It is quite common that we have a variable `x` and want to set it to one value if a condition holds and another otherwise. Suppose for instance, we want `age_group` to be "adult" if another variable `age` is at least 18 and "child" otherwise. As of now, we would write this code.

```
age_group = ""
if age >= 18:
    age_group = "adult"
else:
    age_group = "child"
```

Since this is so common, Python has provided a shortcut.

```
age_group = "adult" if age >= 18 else "child"
```

This code will set `age_group` to "adult" if `age >= 18` and to "child" otherwise (as desired). This is quite useful because it allows us to omit the initialization step of setting `age_group = ""` and reduces our code to one line. In general a ternary operator has the form

```
(true result) if (condition to check) else (false result)
```

Note that there are no colons involved in a ternary operator as with regular `if` and `else` statements.

4 Control Flow II: Loops

In this section, I will discuss the second primary component of controlling the flow of programs: loops.

4.1 While Loops

A while loop can be used to run a body of code while a condition is true. For example, suppose we are given a number $n = 2^k$ and we want to calculate k. We can use a while loop to do this with the following code assuming we have defined some variable `n`.

```
k = 0
while n > 1:
    k += 1
    n /= 2
```

In this loop, the code

```
k += 1
n /= 2
```

will be executed while `n > 1`. Once `n <= 1`, then the code following the loop will be executed. It is important to note that we must declare `k=0` outside of the loop as it is updated at each loop iteration until we reach `n=1`.

For another example, suppose we want to compute a^b. We can do this with the following code

```
result = a
i = 1
while i < b:
    result *= a
    i += 1
```

In this loop, the code

```
result *= a
i += 1
```

will be executed while `i < b`. In the previous code, `n` serves as the iterating variable. We reduce `n` by 2 until it is 1 and count the number of divisions that were required in `k`. In this code, the iterating variable is `i`. We multiply `result` by `a` until `i` has reached `b`. Again, it is important to note that since they are used inside the `while` loop, `result` and `i` are declared outside the loop.

4.2 For Loops

It is very common that we loop from a to b one number at a time. This gave rise to the `for` loop. The `for` loop can be used to provide an alternate implementation for computing a^b.

```
result = a
for i in range(1,b):
    result *= a
```

`range(m,n)` yields a sequence of integers $m, m+1, \ldots, n-1$. The statement `for i in range(1,b)` will then iterate over each `i` in the sequence `1,2,...,b-1`.

It is even more common for you to iterate on a range `0,1,...,n-1`. This can be done with the statement `for i in range(n)`. For example, we can sum the elements from 1 to `n` with the following code.

```
sum = 0
for i in range(n):
    sum += i + 1
```

Due to the `for` loop, `while` loops are generally reserved for checking more complex boolean conditions. For example, it is commonly used for waiting for valid user input.

4.3 Nested Loops

You can also nest loops within each other as a loop can be thought of as any other code to be executed in the body of another loop. For instance, suppose we want to print a 12 times table. That is, we want to print $1*1$, $1*2$, up to $1*12$ and similar for 2 through 12. This can be done with the following code

```
for i in range(1,13):
    for j in range(1,13):
        print(i*j)
```

This will print

```
1
2
...
12
2
4
...
24
3
6
...
36
...
12
24
...
144
```

It is important to note here that since the iterating variable `i` is used by the outer loop, we need a new iterating variable in the inner loop `j`. For each iterating `i` of the outer loop, we run the inner loop from `j=1` to `j=12`. In the innermost loop, we print `i*j`. At first `i=1,j=1` then `i=1,j=2` until `i=1,j=13` and the inner loop ends. Then, we move to the second iteration of the outer loop where we start at `i=2,j=1`, continue to `i=2,j=13` and repeat the process until `i=13` and the outer loop is over.

5 Printing

In this section, I will discuss the `print` function in more detail.

5.1 Printing Multiple Data Types

Until now, we have been printing different data types by just passing them into `print`. For example, in this code we print an integer value, a floating point value, a boolean value, and a string value.

```
print(1)
print(2.3)
print(True)
print("hi")
```

Suppose we have the following variables

```
first_name = "John"
last_name = "Smith"
age = 13
```

and we want to print "John Smith is 13 years old!" using the variables above. We can do this by building a string to be printed

```
pstr = first_name + " " + last_name + " is " + str(age) + " years old!"
```

And then print it to see the desired result

```
print(pstr)
```

However, we do not need to build the result of the string concatenation into a string before printing it. We can just perform the concatenation when calling print:

```
print(first_name + " " + last_name + " is " + str(age) + " years old!")
```

5.2 The Newline Character (\n)

The character \n is a special string character that can be used to separate a string onto multiple lines. For example, suppose we want a string formatted onto two lines that would look like this

```
Hello
World!
```

This string would be stored in a variable like this

```
hw = "Hello\nWorld!"
```

If we then printed `hw`, we would see "Hello" and "World!" on separate lines as above.

5.3 Printing on the Same Line

Note that if we write

```
print(1)
print(2.3)
print(True)
print("hi")
```

That this outputs

```
1
2.3
True
hi
```

Or in other words, each print statement places its input on a newline. Suppose we wanted to place 1 and 2.3 on the same line. To do so, when we print 1, we will need to specify that we don't want the next print to be placed on a new line. This is done by using the `end` parameter in `print()`.

```
print(1, end="")
print(2.3)
```

The output of this code is

```
12.3
```

The parameter `end` specifies what should be printed at the end of the current line after the input you passed in. By default, when you call `print`, `end` can be thought of as already being set to `\n`. By specifying `end` as the empty string `""`, then nothing is printed after 1 which means that the subsequent call `print(2.3)` will place 2.3 directly after 1. Alternatively, suppose we want to print 1 followed by a space, then 2.3. We can do this with the following code

```
print(1, end=" ")
print(2.3)
```

6 Lists

In this section, I will be discussing the list data structure and some of the operations one can perform on it.

6.1 Creating Lists

One can create an empty list `ls` in two ways

```
ls = list()
```

or

```
ls = []
```

You can also create a list with some elements already in it. For instance, suppose you want to create a list with elements 1, 2, and 3 in that order. We can do this with the following code

```
ls = [1, 2, 3]
```

Now, suppose you want to create a list with 10 zeros. Python provides a shortcut to do this using the multiplication operator

```
ls = [0]*10
```

This is equivalent to writing out

```
ls = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

6.2 Indexing a List

Python lists are indexed from 0 to 1 less than the length of the list. The length of a list `ls` can be found using the `len` function. For instance, this code will print 3.

```
ls = [1, 2, 3]
print(len(ls))
```

You can index elements of a list using the `[ ]` operator following the list name. For example `ls[0]` is 1, `ls[1]` is 2, and `ls[2]` is 3. You are able to pass in any value between 0 and `len(ls)-1` into `[ ]`.

You can use `[ ]` to change the element at an index in the list. For example, suppose we want to change

```
ls = [1, 2, 3]
```

to

```
ls = [1, 5, 3]
```

without creating an entirely new list. We can do this with the following code

```
ls[1] = 5
```

We can think of `ls[i]` as a new variable essentially. The following example will illustrate this. Suppose that we have a list `ls` and we want to negate all of the elements in the list. This can be done using the loop

```
for i in range(len(ls)):
    ls[i] *= -1
```

Python provides support for printing a list, so you can call `print(ls)` to see the desired result.

```
[-1, -2, -3]
```

6.3 Adding to a List

You can add new elements to the end of a list using the `append()` function on a list. For example, if I have the list

```
ls = [1, 2, 3]
```

I can add 4 to the end of it by calling

```
ls.append(4)
```

6.4 Two-dimensional Lists

Python, by default, does not have a separate data structure for two dimensional lists (or matrices). We can represent two dimensional lists as a list of lists. Each list can be thought of as a row of a two dimensional list. Suppose I want to represent the 3x3 identity matrix

```
1 0 0
0 1 0
0 0 1
```

This can be done with the following code

```
identity = [[1, 0, 0],[0, 1, 0],[0, 0, 1]]
```

Each of the elements of `identity` is itself a list, as we can see here

```
print(type(identity[0]))
print(type(identity[1]))
print(type(identity[2]))
```

This outputs

```
<class 'list'>
<class 'list'>
<class 'list'>
```

You are allowed to space out the rows of `identity` so it looks more like a matrix

```
identity = [
    [1, 0, 0],
    [0, 1, 0],
    [0, 0, 1]
]
```

Since we know that a two dimensional list is really a list of lists, we will need to index into two different lists to see a particular element. For example, suppose you want to retrieve the upper left element of the identity matrix. We know that this is in the first column of the first row. We can use this logic to identify the two list indexes we have to make. First, we want to get the first row in `identity`. This can be retrieved with `identity[0]`. Now, we want to get the first column in this row. This can be retrieved by getting the first element of `identity[0]` with `identity[0][0]`. Printing `identity[0][0]` displays 1 as expected.

In general, `identity[i][j]` produces the element that is stored in the jth column of the ith row. Applying the logic from above, we first get the ith row with `identity[i]` and then get the jth element of it with `identity[i][j]`. Note that the number of rows in a two dimensional list is

`len(identity)-1`. It is also important to note that when we think of matrices, we assume that each row has the same number of columns. Since a Python two dimensional list is really a list of lists, each of those lists could have different lengths. Therefore, the number of elements (or columns) of ith row is `len(identity[i])`. Therefore, the above indexing only applies for $i = 0, 1, \ldots,$`len(identity)-1` and $j = 0, 1, \ldots,$`len(identity[i])-1`.

If we print a two dimensional list, the output will not look like a matrix. For instance, `print(identity)` will yield

```
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

However, we can use a combination of nested loops and `print()` with different `end` values to print a matrix that looks like

```
1 0 0
0 1 0
0 0 1
```

We need nested loops because we will have an outer loop that iterates over each row and then an inner loop that iterates over each column. We want to print each element in a row on the same line, but keep rows separated.

```
for i in range(3):
    for j in range(3):
        print(identity[i][j],end=" ")
    print("")
```

The outer loop iterates over each row `i`. The inner loop iterates over each column `j` in row `i`. Within the inner loop, we end each print with `" "` as opposed to printing onto a newline. After we print a row, `print("")` will print a newline and nothing else. That way, future prints in the following row will be on a newline. It is important to note that this technically prints a space at the end of each row even though it isn't apparent in the output.

6.5 The `range()` Function (What does `for in` really mean?)

Recall our syntax for writing a `for` loop in Python:

```
for i in range(start,stop):
    ...
```

`range()` is actually a function that produces a list-like sequence of numbers depending on the supplied parameters. The parameters of `range()` are:

- `start` : the start of the sequence of numbers that `range()` produces. If this is not provided, it is set to 0.
- `stop` : the number that is the last number in the sequence that `range()` produces plus `step`. That is, it is not included in the produced sequence.
- `step` : the number that determines the step between the elements of the sequence that `range()` produces. If this is not provided, it is set to 1.

Here are three examples that demonstrate the three common ways this function is used:

- `range(5)` : this produces the sequence $0, 1, 2, 3, 4$. If only one parameter is provided to `range()`, it is interpreted as the `stop` parameter. `start` is 0 and `step` is 1. Note `stop = 5` is not included in the sequence.
- `range(1,4)` : this produces the sequence $1, 2, 3$. Here the `start` and `stop` parameters are provided, and `step` is 1. Again, `stop = 4` is not included in the sequence.
- `range(5,-1,-1)` : this produces the sequence $5, 4, 3, 2, 1, 0$. Here all three parameters are provided. We specify that we want to decrease by having `stop < start` and a `step` of -1. We can see that this is indeed the step of the sequence above. Again, `stop = -1` is not included in the sequence.

It turns out that `for e in s` just iterates over the elements `e` of a sequence `s`. We can see that this is indeed true in the case of `range()` based on the above discussion. In fact, we can use it to just iterate over the elements of a list without explicitly indexing the list. That is, we can now write a `for` loop that mimics "for each element in list l" as opposed to "for each index in list l". For example, suppose we want to write a function that sums the elements of a list `l`. We would originally write this as:

```
def sum_list_original(l):
    s = 0
    for i in range(len(l)):
        s += l[i]
    return s
```

Now that we know `for e in s` iterates over the elements `e` of a sequence `s`, we can use this to define a newer version of this function:

```
def sum_list_new(l):
    s = 0
    for e in l:
        s += e
    return s
```

Now, it is important to note that you cannot replace every loop that goes over a list using indices with one that goes over the elements. To sum all of the elements of a list, the indices are not needed. However, suppose we wanted to write a function to see if two lists `l1,l2` of equal length are equal. We still need to compare the elements at each index of the two lists as follows:

```
def list_equals(l1,l2):
    for i in range(len(l1)):
        if l1[i] != l2[i]:
            return False
    return True
```

7 Mutability

List variables have different properties than variables of primitive types such as integers, booleans, strings, and floating point numbers.

7.1 Idea of a Pointer

In memory, a list variable can be thought of as a pointer to an array of adjacent cells. For example, suppose we create a list

```
ls = [1, 2, 3, 4, 5]
```

The variable `ls`'s memory representation looks something like Figure 1.

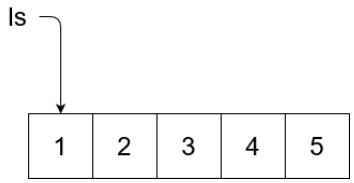

Figure 1: Memory Diagram for `ls`

We can think of `ls` as more simply storing an address in memory where the actual data (1, 2, 3, 4, 5) is stored. The arrow in Figure 1 indicates that `[1, 2, 3, 4, 5]` is not directly associated with `ls` but it is stored at an index known by `ls`. An alternate diagram is shown in Figure 2.

1001	ls = 12001
....	
12001	*1*
12002	*2*
12003	*3*
12004	*4*
12005	*5*

Figure 2: Alternate Memory Diagram for `ls`

In this diagram, the variable `ls` is stored at some memory location labelled 1001. The value of `ls` is actually another memory address labelled 12001. 12001 through 12005 store the 5 elements in the list (these are italicized to convey that they are integer data not other memory locations). When we index `ls` with `ls[i]` Python will go to the location `ls` refers to (12001 in Figure 2) and then move over by `i`. For example, when Python executes `ls[2]`, it will go first to 12001, then move over by 2 to 12003. We can now read or update the data at the memory location 12003.

Therefore, we can see that we are changing the data that is in the list without pointing `ls` to some other memory location. `ls` remains at 12001 while we can change the data in locations 12001 to 12005.

7.2 Tuple, the Immutable List

Python provides an immutable collection known as a tuple. You cannot change the elements of a tuple, which makes it immutable. We can create a tuple using parenthesis as opposed to square brackets.

```
tup = (1,2,3,4,5)
```

We can print its type which is

```
<class 'tuple'>
```

We can access the elements of `tup` in the same way as with a list. For example, `print(tup[1])` prints 2. It is important to note that a single element tuple has special syntax:

```
stup = (1,)
```

That is, you need to place a comma after the one and only element in the tuple. Tuples are generally used to collect grouped information that will not be changed. For instance, we can easily represent a (x, y) point in $\mathbb{R}^2$ as a 2-element tuple.

7.3 Copying Lists and the Shallow Copy

Now that we have introduced that a list is a mutable data structure, what happens when we execute the following code?

```
ls = [1,2,3,4,5]
ls2 = ls
```

Recall Figure 2 on page 15. `ls` is really just a memory location 12001. Therefore, `ls2 = ls` simply copies the value of `ls` into `ls2`. Thus, `ls2` is also 12001. We can represent this in cells in Figure 3, assuming that `ls2` is stored in a memory cell labelled 1002.

1001	ls = 12001
1002	ls2 = 12001
....	
12001	*1*
12002	*2*
12003	*3*
12004	*4*
12005	*5*

Figure 3: Memory Diagram for `ls` and `ls2`

A graphical interpretation of this akin to Figure 1 can be found on the following page in Figure 4. Therefore, we can now see that `ls` and `ls2` point to the same chunk of data in memory. Thus, if we run `ls2[2] = 0` then the data at memory cell 12003 will now be 0. Therefore, when we run `ls[2]`, we will get 0. Updates to `ls2` will now affect the data pointed to by `ls` (as it is the *same data*).

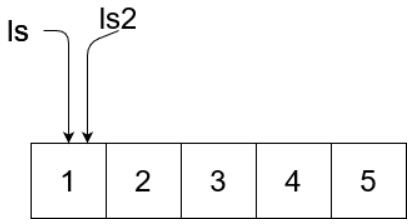

Figure 4: Graphical Memory Diagram for `ls` and `ls2`

What we really want when we copy data is to have something that looks like Figure 5.

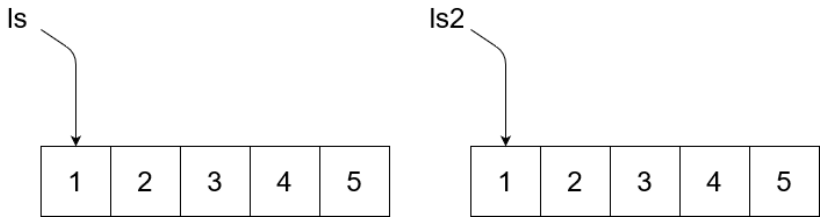

Figure 5: A True Copy of `ls`

That is, we have `ls` and `ls2` both referring to lists with elements `[1,2,3,4,5]` stored separately. An example memory set up is given in Figure 6 which is akin to figures 2 and 3.

Now, suppose we execute the following code. What will it output?

```
ls2[2] = 0
print(ls[2])
```

This will output 3. That is, the data pointed to by `ls` was not updated. Why is this? Remember, when we call `ls2[2] = 0`, Python will go to 22001 (see Figure 6), move down by 2 to 22003, and update the data at cell 22003 to 0. Then, when we run `print(ls[2])`, we go to 12001, move down by 2 to 12003 and print the data at cell 12003 which is 3. Note that the memory addresses of `ls` and `ls2` (12001 and 22001 respectively) were chosen for no particular reason. When we perform a copy akin to figure 4, we call this a *shallow copy*. When we have two separate lists as in Figure 5, we call it a *deep copy*.

1001	ls = 12001
1002	ls2 = 22001
....	
12001	*1*
12002	*2*
12003	*3*
12004	*4*
12005	*5*
....	
22001	*1*
22002	*2*
22003	*3*
22004	*4*
22005	*5*

Figure 6: A True Copy of `ls`

7.4 What about Primitives? Are they Mutable?

Primitives are variables of primitive data types. That is, of types `bool`, `int`, or `float`. For example,

```
int_var = 5
float_var = 3.14
bool_var = True
```

When we ask the program for `int_var`, we just get back the integer stored at some memory location. In essence, it is not a pointer that points to some cell that can be edited. It is the same for `float` and `bool` type variables as well. Suppose we wrote the following code.

```
int_var = 5
int_var2 = int_var
```

This will copy the contents of `int_var` into `int_var2`. That is, they will both be 5. The only way we can change `int_var2` is to assign it to something else which will have no effect on `int_var`.

8 Functions

In this section, I will cover the purpose and components of functions, different types of functions, and how they will be invoked (or called).

8.1 Why do we need functions? (Introduction to Modularity)

In math, a function takes an input from some domain and returns an output in some range. We can create the equivalent of various mathematical functions. For instance, we could create the equivalent of $y = f(x) = 3x + 5$.

```
def y(x):
    return 3*x + 5
```

The Python keyword `def` creates a new function named `y` that takes a single input `x`. The Python keyword `return` causes the function to end and outputs the result of the expression that follows it (in this case the expression `3*x + 5` evaluated at whatever `x` is). We can retrieve `y` values that correspond to various `x` by invoking or calling `y(x)`. For instance, `print(y(3))` prints 14.

We can also create functions with multiple parameters. For instance, suppose we wanted to create the equivalent of the mathematical function $z = f(x, y) = 3x + 2y + 5$. This can be done with the following code.

```
def z(x,y):
    return 3*x + 2*y + 5
```

This function has two parameters `x` and `y`. We can call it with 2 parameters and present the result. For instance, `print(z(3,7))` prints 28.

In programs, we can think of a function as any kind of action while variables correspond to quantities. This is because functions in programs do not have to take input and do not have to return output. Functions can edit our input or can display output to the user via the console or a graphical user interface.

We could technically implement our programs without any functions. However, by breaking up all the actions required to implement a larger program into functions, it will make our programs easier to

read and debug. We can test functions separately and make sure they work before incorporating them together. This is known as *unit testing*. By thinking of functions as the equivalent of actions, we can write the larger program assuming that the actions are already implemented. For instance, suppose we wanted to write a program that repeatedly prompts a user for information on a person (name, age, gender, etc.) and then adds that information to a database. We could come up with an algorithm in English that is something like

```
while (the user is not done)
    (prompt for person info)
    (process the info into the appropriate data types)
    (upload info to database)
    (confirm the upload)
```

Each of the statements in parenthesis are actions that we will need to perform. Aside from that, the only thing left in our algorithm is the "while" phrase which we can replace with a Python `while` loop. If we didn't use functions, our algorithm would not be so simple as we would have to write out all of these steps in one (very long) `while` loop. By implementing our actions as separate functions, we can not only test them separately, but also retain our simple algorithm idea. The idea of breaking up our programs into separate functions is known as *modularity*.

8.2 Parameters v. Arguments

The parameters of a function are what follow the `def` keyword and are enclosed in parenthesis. We write the body of the function in terms of the parameters (just like a mathematical function). Arguments are the values that are substituted in for the parameters when a function is invoked or called. For instance, when we call `y(3)`, 3 is an argument being substituted for the parameter `x`.

The arguments to a function can also be variables. For instance, suppose we wanted to evaluate `y(x)` for 1 to 10 using a `for` loop and print them.

```
for xi in range(1,11):
    print(y(xi))
```

Note here that the argument when calling `y` is itself a variable `xi`. When `y(xi)` is executed, Python creates a *copy* of `xi` and passes it as the argument to `y`. One could ask, based on the discussion of the *shallow copy* in section 7, what is meant by a copy here? When copying a variable as an argument, Python will use the `=` assignment operator, which will perform a shallow copy in the case of a list (and other mutable data structures). That means, since the argument is a shallow copy of a list that exists outside of the function, we can edit the argument. This is discussed in more detail in section 8.5.

8.3 Variables inside a Function (and Scope)

In the example functions `y(x)` and `z(x,y)`, we just returned an expression that is in terms of the parameters. However, we can indeed create variables inside a function. For instance, suppose we wanted to write a function that is the equivalent of the third degree polynomial $y = f(x) = 5x^3 + 4x^2 + 3x + 6$. We can do this with the following code which mimics the definitions of `y(x)` and `z(x,y)` above.

```
def yp(x):
    return 5*x**3 + 4*x**2 + 3*x + 6
```

Note here that in Python `**` is used to raise a number to a power. For instance, `x**3` is the equivalent of x^3. However, we could alternatively implement it by making a variable for each polynomial term and then sum them with the y-intercept 6.

```
def yp2(x):
    cube_term = 5*x**3
    quad_term = 4*x**2
    lin_term = 3*x
    return cube_term + quad_term + lin_term + 6
```

The variables `cube_term`, `quad_term`, and `lin_term` are known as *local* variables to the function. They only exist during function execution and *cannot* be accessed outside the function. Local variables expressed in terms of the parameters can be used to store steps in building the function output or to serve as iterating variables in loops.

In fact, variables declared inside a loop behave a similar way. Variables declared inside a loop will be reset at each iteration. Their state will not be saved between iterations. Similarly, variables in a function do not retain their state between function calls. The block of code where a variable's state is retained is known as its *scope*.

8.4 The Return Statement (How many do we need?)

`return` is a reserved Python keyword that terminates the execution of a function once it is executed and returns the result of the expression that follows it to the calling code. For example, suppose we wanted to write the equivalent of the piecewise mathematical function

$$f(n) = \begin{cases} \sum_{i=1}^{n} i & \text{if } n \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

This can be written in Python as.

```
def psum(n):
    if n < 1:
        return 0

    s = 0
    for i in range(1,n+1):
        s += i
    return s
```

There are multiple levels of indentation in this function. All lines in the body of the function must be indented by one over from `def`. Thus, the lines of the `if` statement and `for` loop will be indented by 2 from `def`. We are trying to implement a piecewise function, so we know that there are 2 different quantities that could be returned. The summation from 1 to n if $n \geq 1$ and 0 otherwise. This implementation of `psum()` was written with this idea in mind. If `n < 1`, we can immediately return 0. Otherwise, we need to compute the sum `s` before returning it.

Just because this is a piecewise function with two cases, we do not need two return statements. We could alternatively write this function as

```
def psum2(n):
    s = 0
    if n >= 1:
        for i in range(1,n+1):
            s += i
    return s
```

In this version of the function, we start the sum `s` at 0 and only update it to be the sum from 1 to n if $n \geq 1$. We return whatever `s` ends up being based on the if check.

The stylistic differences between these two implementations can be extended to more general function. The first version deals with simple outputs first. That is, it deals with outputs that can be computed directly before moving on to more complex outputs. The second version will compute the more complicated output only if the requirements for computation ($n \geq 1$ in this case) are met.

In general, you should prepare the output in whatever way you please (based on some conditions), and `return` whenever it is ready noting that a `return` can output an expression by evaluating it first.

Now, a function does not need to return anything. For instance, we could have a function that just prints something (which displays output to the user via the console). Python will by default return `None` if a function is missing a `return` statement. `None` is a Python reserved keyword that stands for nothing or *null*.

8.5 Functions that Edit Input

I mentioned in section 8.2 that if we have a function that takes a list as a parameter, it can in fact edit the list because of how arguments are copied before being passed into the function for evaluation. For example, suppose we have a function that prints the elements of a list on separate lines.

```
def print_list(ls):
    for i in range(len(ls)):
        print(ls[i])
```

And then we execute the following code

```
l1 = [1,2,3]
```

The memory will look something like this

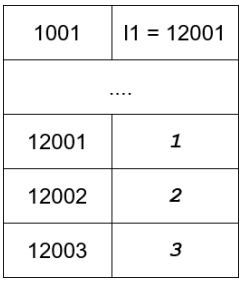

Figure 7: Memory before call to `print_list()`

Python will copy `l1` into `ls` to be the argument of `print_list()` using `=`. Using this information, we can create a memory diagram of the existing variables at the beginning of the function.
Now that the function has been called, `ls` is no longer just a parameter to write the function `print_list()` in terms of, but will now be copy of `l1` created via `=`. Thus, `ls` and `l1` will both point to 12001. Therefore, changing `ls[i]` will affect `l1`, a list that exists outside of the scope of the function `print_list()`. `print_list()` is not a function that edits the input list as it just prints the list, but I wanted to point out the relationship between an argument that can be referenced by `ls` and the list that was copied from the calling code `l1`.

1001	l1 = 12001
1002	ls = 12001
....	
12001	*1*
12002	*2*
12003	*3*

Figure 8: Memory at start of `print_list()`

Now, let's write a function that will negate all of the elements *in* the input list. It will not return a new list.

```
def negate_list(ls):
    for i in range(len(ls)):
        ls[i] *= -1
```

This function will set each element in the argument list `ls` to be the product of its original value with -1. Suppose we execute the following code.

```
l1 = [1,2,3]
negate_list(l1)
```

Using Figure 8, `ls[0] = -1*ls[0] = -1` will really be setting the data at memory location 12001 to -1. Similarly, `ls[1] = -2` and `ls[2] = -3` will be setting the data at memory location 12002 and 12003 to -2 and -3 respectively. This is shown in Figure 9.

1001	l1 = 12001
1002	ls = 12001
....	
12001	*-1*
12002	*-2*
12003	*-3*

Figure 9: Memory after `negate_list()`

Now, we can examine the contents of `l1` using Figure 9. `l1` still points to the data at location 12001. Thus, we can see that `l1[0]` is the data at 12001 + 0 which is -1. Similarly `l1[1]` and `l1[2]` is the data at 12001 + 1 and 12002 which is -2 and -3 respectively. Thus, we can see that the function `negate_list()` actually edits the input argument `l1` and can see this by printing `l1`.

In fact, we can also `append` to an input list and see the results after the function call in a similar manner. The reason as to why this is true is beyond the scope of this document.

8.6 Invoking a Function

For a function that returns no output and does not edit the input, such as `print_list()`, we just call it and see the result on the console.

```
l1 = [1,2,3]
print_list(l1)
```

If we have a function that returns something (but does not edit the input), we can store the result of a function in a variable or use it directly. For instance, consider the function `y(x)`. We can assign the result of `y` being evaluated at some value to a variable.

```
out = y(3)
```

Or, we can just use the result directly as it is an expression (in particular it's the expression `3*x + 5` as noted in section 8.1) that will be evaluated by Python. For example, it can be printed directly

```
print(y(3))
```

It can also be used in a boolean condition like any other expression.

For functions that edit an input, we can call the function on the input and then examine the input (as in the case of `negate_input()` above) afterwards.

Note, we can also have functions that both edit an input and return an output, in which case we would do a combination of the above.

8.7 Returning Multiple Items

In Python, you can actually return multiple values from a function. For instance, suppose we want to implement a function `roots` that returns the positive and negative roots of a number n. Using our knowledge of tuples, we could implement it like so

```
def roots(n):
    rt = n**0.5
    return (rt, -1*rt)
```

If we then run

```
out = roots(25)
print(type(out))
print(out)
```

We see the output

```
<class 'tuple'>
(5.0, -5.0)
```

However, we could also write

```
def roots2(n):
    rt = n**0.5
    return rt, -1*rt
```

Then, we can write

```
rp, rn = roots2(25)
print(rp, rn)
```

As we can see here, `roots2` has 2 separate returned values that can be stored in `rp` and `rn`.

9 Reading User Input

In this section, I will discuss how to read user input from the keyboard in console/command line.

9.1 The `input()` Function

Python provides the function `input()` which takes a string and returns a string. The string that you pass into it will be displayed to the user as a prompt. The string that is returned is the collection of characters the user entered before they hit the enter/return key. For instance, consider the following code.

```
name_in = input("Enter your name! ")
```

This code will display "Enter your name! " on the console and will wait until the user enters some characters and presses the enter key. Then, once the user presses enter, whatever the user input before the enter key will be stored as a string in `name_in`.

It is important to note that since `input()` returns strings, if we have a program that needs to work with numbers, we will have to make appropriate casts (`int()` or `float()`) for instance.

9.2 String to List (`split`)

A useful function when dealing with user input is the string `split()` function. For example, suppose we wanted to write a program that asks the user to enter their first name, last name, age, and gender separated by commas. We could prompt and collect this information with the `input()` function:

```
info = input("Enter your first name, last name, age, and gender separated by commas: ")
```

This will collect the input information into a single string. However, we are more interested in the individual components (first name, last name, age, and gender). Python provides a function `split()` that will convert a string into a list. `split()` takes a string as a parameter that tells us which string we want to "split on". In this example, we have told the user to separate (or split) the data with commas. Therefore, we can break up `info` using this code.

```
info_list = info.split(",")
```

For instance, suppose we entered "John,Smith,20,male" on the keyboard. This string is stored in `info`. Then, splitting on "," into `info_list` yields `info_list` containing

```
['John', 'Smith', '20', 'male']
```

Here we see that we have split up `info` into the four strings that are separated by the three commas in `info`.

10 Organizing Code

In this section, I will discuss some important aspects of organizing code and how it can be implemented in Python.

10.1 Modularity II (Libraries)

In section 8.1, I introduced the idea of modularity as the concept of breaking up our program into independent functions. This can be extended further into splitting up our functions into separate files.

A *library* can be thought of as a collection of general functions. The program that will be run is typically referred to as the *client code*. The client code will use the library functions to perform a more specific application. The client code file is the file you will be running as your program. It will contain a script (code that is not in a function, but will most likely call functions), that will run when you run the file in a Python IDE. It can also contain functions as necessary.

10.2 Importing Files

In Python, all files including library and client code will have the extension `.py`. Suppose that your client code is in `main.py` and you have a library of functions `math_library.py` that looks like:

```
def y(x):
    return 3*x + 5

def z(x,y):
    return 3*x + 2*y + 5
```

You can import the entire library into `main.py` by adding the following line to the top of the file.

```
import math_library
```

You can then use the functions from `math_library` in `main.py` in the following manner

```
math_library.y(3)
math_library.z(2,3)
```

We need to precede `y` and `z` with math_library or Python will think that these functions are defined in `math.py`. You can import a particular function (such as `y(x)`) from a file using `from-import` syntax.

```
from math_library import y
```

You can import multiple functions separated by a comma

```
from math_library import y,z
```

If you have a library file with a particularly long name, Python allows you to provide it with an alias when importing it.

```
import math_library as ml
```

Then, you can just use `ml` where you would have used `math_library` in `main.py` in the first example.

```
ml.y(3)
ml.z(2,3)
```

10.3 Constants II (in Libraries)

In section 1.4, I mentioned that constants are variables that are not meant to be changed. In Python, there is no way to enforce this, but by naming a variable entirely in uppercase characters conveys to other programmers that these are constants that are not meant to be changed. In math, there is a set of common constants (e, π, etc.). In programs, we use constants quite frequently. Most importantly, constants are used to avoid "magic numbers". Magic numbers are any numbers that appear in your program without explanation.

For instance, suppose you had an environment represented by a two dimensional list. Entries in the 2D list that are 1 represent roads and entries that are 0 represent areas not suitable for travel. For example, we could have an environment that looked like this

```
[
    [0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0],
    [1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0]
]
```

We could initialize this environment using the following code

```
environment = []
for i in range(3):
    environment.append([0, 0, 0, 1, 0, 0, 0])
environment.append([1]*7)
for i in range(3):
    environment.append([0]*7)
print(environment)
```

For this program, I assume that in future we will be moving some entities through the environment and will have to check whether it is travelling on safe ground or not. Or in other words, check if their current position is a 0 or 1. It is better to consider the values in the environment as safe or unsafe. Therefore, it is more appropriate to define constants

```
SAFE = 1
UNSAFE = 0
```

Then, edit our environment creation code

```
for i in range(3):
    environment.append([UNSAFE, UNSAFE, UNSAFE, SAFE, UNSAFE, UNSAFE, UNSAFE])
environment.append([SAFE]*7)
for i in range(3):
    environment.append([UNSAFE]*7)
```

In future code, when we have entities navigating the environment we can check positions to be `SAFE` or `UNSAFE`. While this is equivalent to just using 0 and 1, it produces more readable code and allows us to change the representation of safe and unsafe areas in the future.

The concept of magic numbers applies to other data types. We want to avoid magic strings as well.