Python Programming

Technical Document

CS1030

Jimmy Torres

Table of Contents

Variables, Data types, and Methods

Variables

a.) A variable is a piece of data/value. A variable is assigned with a single, = , and anything on the right of this is assigned to the variable. Different data types can be assigned such as, a **string (" "/' '), integer(int), float, function** outputs and **Boolean(True or false). Lists** can also be assigned to variables and use brackets. All of these data types are saved on the local device's resources(memory) when the program is running.

Data Types

b.) Strings appear as they are. With a matching pair of single or double quotes at the beginning and end of the item, the string of characters are defined. Numbers that are portrayed as strings such as "3", cannot be used to calculate problems.

Strings may be able to be formatted when they are printed to an output. With the example of "3" in the previous paragraph, a person's age may be represented by a variable called **person_age** and assigned a value of "3". With another variable, a user can "add" the person_age variable to the current variable to make a new string. Below is an example.

Code:

```
person_age = "3"
toddler_info = "The toddler that was " + person_age

print(toddler_info)
```

Output:

```
The toddler that was 3
```

Integers are whole numbers and are used to calculate math problems. Integers cannot be used with floating numbers unless they are converted to the appropriate data type. Integers are defined with a parenthesis and 'int'. `variable = int(3)` is an example of this.

Code:

```
number_one = int(12)
number_two = int(18)

sum_of_numbers = number_one + number_two

print (f"The sum of the numbers is {sum_of_numbers}.")
```

Output:

```
The sum of the numbers is 30.
```

Floats are used to indicate a floating number. A floating number is used to display decimal values up to 32 bits. All float values always have a decimal value. Pi is equal to 3.14159265359… and is an example of a float. Currency requires float numbers or it will not show decimal/cent value.

Code:

```
number_one = float(12.77)
number_two = float(18.27)

sum_of_numbers = number_one + number_two

print (f"The sum of the numbers is {sum_of_numbers}.")
```

Output:

```
The sum of the numbers is 31.04.
```

Floats may require instances where rounding is necessary. If a value needs rounding, the **round** method needs to be used with an argument of **2** as the last number. The following example is used with the value of pi.

Code:

```
pi = float(3.1415926535)
rounded_pi = round(pi, 2)

print (f"Pi rounded to 2 decimal places is, {rounded_pi}.")
```

Output:

```
Pi rounded to 2 decimal places is, 3.14.
```

In specific circumstances, an integer may need to be converted to a float, or vice versa. To do this, a new variable must come after the original variable annotating what it's new datatype is.

Code example of float to integer:

```
float_number = float(6.12345)
int_number = int(float_number)

print (f"Original float was 6.12345 and \nis now an integer data type {int_number}.")
```

Output:

```
Original float was 6.12345 and
is now an integer data type 6.
```

Code example of integer to float:

```
int_number = int(7)
float_number = float(int_number)


print (f"Original float was 6.12345 and \nis now an integer data type {float_number}."
```

Output:

```
Original float was 7 and
is now an integer data type 7.0.
```

Booleans are used to identify true or false statements. An example of this is using any of the following relational operators to compare values in a Boolean statement:

< — Less than

> — Greater than

== — Equal to (can also be used with string values)

<= — Less than or equal to

>= — Greater than or equal to

!= — Not equal to (can also be used with string values)

Using the print method to visualize this logic, a non-string comparison of numbers can be compared and the computer can interpret the information. The following example compares the value of 1 to another 1.

Code:

```
print (1>1)
print (1==1)
print (1<1)
```

Output:

```
False
True
False
```

In short, **1 is equal to 1**, and any other comparison is **False**. When a user is not using the print method, this may be used in conditional loops such as, **while True** loops, and **if** statements. Review the loops section in this document to view examples of practical uses of this logic.

Lists are used to either have several values saved in the using device's memory, or store future inputs or new values that become calculated later. List variables use a matching pair of brackets, **[ ]** , to contain items. The below example shows string values saved in a variable called list_example. Lists are used to store single items, several items, or no items at all.

Example of a list variable being used with several list items:

```
list_example = ['item_one', 'item_two', 'item_three']

for items in list_example:
    print(items)
```

Output:

```
item_one
item_two
item_three
```

Using the **print** method for a list without a **for** loop may cause the list to print exactly the way it appears in the code.

Example of using **print** without a **for** loop:

```
list_example = ['item_one', 'item_two', 'item_three']

print(list_example)
```

Output:

```
['item_one', 'item_two', 'item_three']
```

The reason for this is that the computer does not know how the user wants the information to be displayed. A **for** loop works line-by-line through each **list** item until there are no items left. The result is a printed list that appears as a column.

Example of a **list** variable with **no values** assigned:

```
userinputs = []

def input_function():

    while True:
        userinput = input("Please enter an input to save, type 'done' when you are finished:\n")

        if userinput == "done":
            break
        else:
            userinputs.append(userinput)

input_function()

for userinput in userinputs:
    print (userinput)
```

Output:

```
Please enter an input to save, type 'done' when you are finished:
rock
Please enter an input to save, type 'done' when you are finished:
ball
Please enter an input to save, type 'done' when you are finished:
box
Please enter an input to save, type 'done' when you are finished:
done
rock
ball
box
```

The **red boxed** variable is an empty list. The function in the **yellow box** allows the user to type and enter **inputs** as **string**s. The **userinput** variable asks for a value be entered. As long as the user does not type **'done'**, the value entered will be added to the ***userinputs list***. This is done using the method named, **userinputs.append(userinput)**.

Explanation of **list** items in **green** boxes:

Userinput is a variable that only takes one input at a time. Using the **userinputs.append** method, items are added to the ***userinputs*** list.

With an argument of **(userinput)**, the program knows to add the userinput variable to the ***userinputs*** list. When the loop restarts, the process starts over and the previous userinput is changed, and then added to the list of **userinputs** that have been accumulating. The only way to **break** the loop is to type, **'done'**.

Methods

c.) A method is a function. Unlike a variable, a method can pass information (arguments). Some methods return information.

Print is a **method**. A method sometimes needs an argument, and is annotated in parenthesis or attached to a variable name. Using **print** to display a string, the following example shows how print is used with parentheses and double quotes at the beginning and end of the string:

Code:

```
print("This line of text is printed")
```

Output:

```
This line of text is printed
```

Print can also use **formatting** to print existing variables. Using an **f** inside the argument, but outside the quotation marks, a user can add variables to print. Using curly brackets, a user can name the variables to display in the output. Below is an example of **print** being used with formatting:

Code:

```
a = 1
b = 2
c = (a) + (b)


print(f"the sum is {c}")
```

Output:

```
the sum is 3
```

Other examples of methods such as **round** , **.append** , **.lower** , **.upper** , **.join** , are examples of methods to make an action occur. Methods depending on their usage can manipulate information and create new outputs for other parts of a code.

Comments

d.) **Comments** are used to make notes about the code and are hidden from the viewer. The following types of comments can be used in a python file:

Multi-line comments are used by using a beginning and end tag of three consecutive single or double quotes. All characters in between these tags will be a comment and become highlighted as such. This type of comment can span several lines if needed.

Example:

```
'''
This is a
multi-line
comment
'''
```

```
"""
This is also a
mult-line comment
"""
```

Single line comments use one hashtag/number sign at the beginning of the comment. These are highlighted differently than multi-line comments and only span to one line per comment. Several lines of comments require a hashtag/number sign at the beginning of each line.

Example:

```
# This is a single line comment.
# This is another.
```

Pseudocode and Flow Charts

a.) Pseudocode is used to outline a program or code. This method of abstraction is essentially a draft of how the hierarchy of a code will work and also how the logic is supposed to be represented.

The purpose of this is to foresee any possible issues or improvements that can be made before building code in the compiler. This is comparable to making a rough draft for how a house will be built before attempting to build the house with the materials. Doing this can prepare the user and develop an understanding before typing and compiling code.

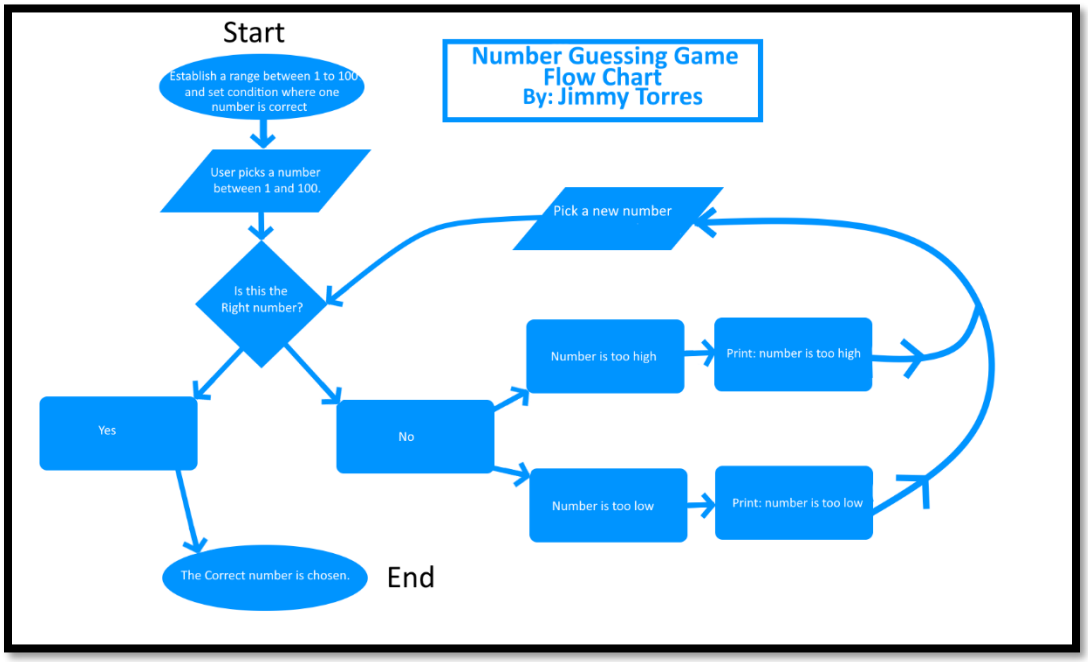
Below is an example of basic pseudocode:

While this example is a crude example, typing on text file fulfills the same purpose.

b.) Flow charts are another type of abstraction that gives a visual representation of how a code is supposed to work. The topology of a flow chart should have a **Start** and an **End,** to indicated where a program begins and stops. Symbols in flow charts represent certain points where code is executed and have the following definitions:

- **Circle/oval** – Start and stop. This represents the first and last piece of code in the program. The Starting code initializes the program.
- **Arrows** – Indicates where the next process will be. This is the **'flow'**. Crossed lines in crowded flow charts may have bent/curved lines to indicated they are 'jumping' over unrelated processes.
- **Rectangle/square** – This indicates a process.
- **Rhombus** – This represents an input by the user.
- **Diamond** – This represents a decision. If a specific condition is met, it is processed to the next line of code. This is not an input.

Example of a flow chart for Guessing Number Game:



Selections and Inputs

Reading Input

a.) Input is what the user is putting into an application or code. This document is typed with a keyboard and every button pressed is an input. It is the computer's job to make an output, which in this case is the text appearing on this document.

With python, a code or program may be written to receive input before making further outputs. This may be any type of data type needed to continue using the program. Below is an example of a user being asked to enter their favorite color.

Code:

```
favorite_color = input("Please enter your favorite color")

print(f"The user's favorite color is {favorite_color}")
```

Output:

```
Please enter your favorite colorblack
The user's favorite color is black
```

The **favorite_color** variable was empty before the input 'black' was entered. Because the variable was changed to the value of 'black', the **print** line of code had used the newly assigned value for the output.

This same method can be used for calculating unknown math problems. This example turns the input string into an integer, then the program can interpret the information correctly.

Code:

```
a = int(input("enter a number to be added to 2 : "))
b = 2

c = (a) + (b)

print(f"{a} + 2 = {c}")
```

Output:

```
enter a number to be added to 2 : 3
3 + 2 = 5
```

Conditions

b.) Conditions are used to ensure a specific set of parameters are met. This requires a loop in order to cycle through the different possibilities. In a scenario where a loop is continuously not getting the correct input, it go on indefinitely.

The **if** condition can be set up to meet certain conditions. Building on the last example of using **input**, this line of code takes an input to be added. If the result is equal to (**==**) 5, it will print a string to tell the user they are right. Any other result (‘**else**’) results in a statement that indicates they are wrong.

Code:

```
a = int(input("We are trying to add up to 5 \nEnter a number to be added to 2 : "))
b = 2

c = (a) + (b)

if c == 5:
    print ("That's right!")
else:
    print("Not quite...")
```

Output #1:

```
We are trying to add up to 5
Enter a number to be added to 2 : 3
That's right!
```

In this circumstance, the **condition** that was **equal** to 5 was met“That’s right!”, was printed as a result.

Output #2:

```
We are trying to add up to 5
Enter a number to be added to 2 : 2
Not quite...
```

In this circumstance, the input did not equal 5 and the **else** condition printed a different string.

The next section gives 2 other examples with more detail about **if**, **elif**, **else**, in a **while True** loop.

Loops and Algorithms

While Loops

a.) **while** loops are used to indicate that while a certain variable or condition is **true** or **false**, a set of conditions are to occur. Including '**while True:**' to the beginning of a condition can make the conditional statement into a loop. The purpose of this is to force a specific set of parameters be met, in order to make use of the input or output of a piece of code.

The issue with **while** loops is if they are incorrectly written, they continue to run through the section of code indefinitely (or until the program crashes). To prevent this, the following needs to be implemented:

- The variable requiring input (if any) must be added after the **while True:** loop starts, and BEFORE the variable that will be **true** or **false**.
- The variable being checked for **true** or **false** must be placed before the conditional statements start (**if**, **if not** , etc.).
- The first condition is always **if** (or **if not**) and is followed by a relational operator (see example).
    - The **if** statement requires that something happens (what happens if condition is met).
- **elif** is always the next conditional statement. It is formatted in the same way that, **if**, is formatted but must either have a different relational operator, or it must have a different condition that is met. Contradicting statements will cause errors.
- **else** refers to all other circumstances of comparing that are not applied. Else applies to every condition that is not already listed. If an error occurs with this condition, there is a syntax error (such as a string being entered as an input, when an integer is required).
- In the event that **else** is used, it should be used in the last condition of the loop.
- There must be a '**break**', or a call for a **function**, to stop the loop.

Code for a while True using only **elif**:

```
b = 2

while True:
    a = int(input("We are trying to add up to 5 \nEnter a number to be added to 2 : "))
    c = (a) + (b)
    if c == 5:
        print ("That's right!")
        break
    elif c < 5:
        print ("Too low! try again \n")
    elif c > 5:
        print("Too high! try agin \n")
```

Output:

```
We are trying to add up to 5
Enter a number to be added to 2 : 1
Too low! try again

We are trying to add up to 5
Enter a number to be added to 2 : 10
Too high! try agin

We are trying to add up to 5
Enter a number to be added to 2 : 3
That's right!
```

Code for a while True using **else** instead of elif:

```
b = 2

while True:
    a = int(input("We are trying to add up to 5 \nEnter a number to be added to 2 : "))
    c = (a) + (b)
    if c == 5:
        print ("That's right!")
        break
    elif c < 5:
        print ("Too low! try again \n")
    else:
        print("Too high! try agin \n")
```

Output (same output as the other example):

```
We are trying to add up to 5
Enter a number to be added to 2 : 1
Too low! try again

We are trying to add up to 5
Enter a number to be added to 2 : 10
Too high! try agin

We are trying to add up to 5
Enter a number to be added to 2 : 3
That's right!
```

The previous 2 examples show that the output of the 'c' variable needs to be equal to '5' to break the loop. The conditions without a '**break**' keep the loop continuing until the input completes the sum. When the user enters an integer value of '3', the loop stops. If '**break**' is removed from the if condition, the loop is never completed.

for Loops

b.) For loops are used to repeat tasks for a set amount of times. **for** loops are useful giving outputs to the user. A **for** loop consists of a **list** or **dictionary**, an **argument** with **integers**, and a method such as **print**.

A **for** loop can be used in calculating math problems that may repeat several times (see example) and printing series of strings.

In this example, **for** starts the loop. **i** is the name given to the items that will be printed. Any name can be given to these items and the way this is assigned is similar to a variable being assigned a name. **in range** is specific to going through a specific range of numbers in the **argument / ()**.

The **argument**'s first number is **1**. This number indicates that the range starts counting at 1. It is important to note that python always starts counting at **0**. This means if a user wants a range of 9, it will stop at 8, because it started counting at 0. The reason for this is computers count in binary, and values range from 0 to 9 before beginning to reuse these values for larger numbers.

The second number in the **argument** is the user input, which will set the max number to whatever the user wants.

The third number in the **argument** indicates that the range will start counting with an offset number of **+1** from 0. This means it will start counting at **1** and not **0**. This is done to not confuse the user when the output is one less number than they had entered.

The multi_table variable is the calculation that will be repeated based on the user's input. For every iteration, **i** will gain a value of **1**. With this in mind, it will be used to multiply the variable in the multiplication problem until the

range's highest number is met. **print** follows this line of code as the final step in this loop. When the loop runs, this **algorithm** repeats the process until the **range** is complete.

Code of a **for** loop used to create a multiplication table.

```
user_input = int(input("Please enter the highest number for multiplacation table for the number three"))

for i in range (1,user_input +1):
    multi_table = 3*(i)
    print(f"3*{i} = {multi_table}")
```

Output:

```
Please enter the highest number for multiplacation table for the number three9
3*1 = 3
3*2 = 6
3*3 = 9
3*4 = 12
3*5 = 15
3*6 = 18
3*7 = 21
3*8 = 24
3*9 = 27
```

Code of a **list** being printed in order of when the items were added:

```
userinputs = []

def input_function():

    while True:
        userinput = input("Please enter an input to save, type 'done' when you are finished")

        if userinput == "done":
            break
        else:
            userinputs.append(userinput)

input_function()

for userinput in userinputs:
    print (userinput)
```

Output:

```
Please enter an input to save, type 'done' when you are finishedball
Please enter an input to save, type 'done' when you are finishedball2
Please enter an input to save, type 'done' when you are finishedball3
Please enter an input to save, type 'done' when you are finisheddone
ball
ball2
ball3
```

Algorithms

c.) Algorithms are a series of steps toward a specific objective. PEMDAS is an algorithm used in math toward solving long math problems. The same applies for Rubik's cube solving, as a user will need to understand steps toward completing layers of colors before finishing the last side of the cube.

When writing functions, loops, etc., a user is developing an algorithm (set of steps) that the program will follow toward giving an output. In python, the higher the line of code is in the **hierarchy**, the sooner it will be executed in the output.

To create a more flexible program, a user can make blocks of code that don't execute until it is **called**. A **function** follows this circumstance. Functions allow a user to call a section of code only when it's necessary to use. Functions can call other functions if the user decides, and make it easier to organize sophisticated code.

Functions require that they be **defined**. Defining a function is straight-forward, as they only contain code that function will execute.

```
def math_table():
    for i in range (1,user_input +1):
        multi_table = 3*(i)
        print(f"3*{i} = {multi_table}")
```

This function only has a **for** loop and will not be executed unless **math_table()** is called somewhere in the code. The example shown has a **variable** in the **for** loop that is not defined in the **function**.

Variables outside of a function can be called from inside the function.

Code:

```
user_input = int(input("Please enter the h

def math_table():
    for i in range (1,user_input +1):
        multi_table = 3*(i)
        print(f"3*{i} = {multi_table}")

math_table()
```

Output:

```
Please enter the highest number for multiplacation table for the number three5
3*1 = 3
3*2 = 6
3*3 = 9
3*4 = 12
3*5 = 15
```

However, a variable created inside a function requires two additional steps. The user needs to **return** the variable at the end of the function. Second, the same variable needs to be created outside of the function with it assigned to the function name where it originated from.

Example:

```
def input_function():
    user_input = int(input("Please enter the highest number for multiplic
    return user_input

user_input = input_function()

for i in range (1,user_input +1):
    multi_table = 3*(i)
    print(f"3*{i} = {multi_table}")
```

In more complex functions, a user may need to use an **input** across several functions. The following example shows a variable called, **'zero'** with an integer value of **0** being passed through two functions and into a separate **for** loop. The following steps are taken:

- **Return** the **input** variable at the end of the first function.
- Naming the variable **outside** of the function with the original function assigned to it.
- **Add the variable as an argument to the next function that will use it. *This step carries the variable to the next function so it can be used.***

At this point a new variable called **sum_didnt_change** uses the carried variable **zero** for a calculation.

- Return the new variable at the end of the function.
- Name the variable **outside** of the function with the name of it's corresponding function assigned to it.
- Use the **variable** where it's needed.

All the repetitive lines of code are highlighted below. Note that with **zero** and **sum_didnt_change** variables, a pattern is emerging for passing the values outside of their functions.

Example of a variable being passed on to other functions:

```
def useless_function():
    zero= int(input("please enter the number zero"))
    return zero
zero = useless_function()

def input_function(zero):
    user_input = int(input("Please enter the highest numbe
    sum_didnt_change = user_input + zero
    return sum_didnt_change

sum_didnt_change = input_function(zero)

for i in range (1,sum_didnt_change +1):
    multi_table = 3*(i)
    print(f"3*{i} = {multi_table}")
```

The following program is made for playing rock, paper, scissors against a computer. The program is designed to work out of the **introduction()** function (Fig. 1).

Fig. 1:

```
playername = input("What username would you like to use? Please enter
print(f"Hi {playername}!")

#The introduction function is what the player is introduced to before
def introduction():
    #while loop sets condition for a specific string that the the loo
    # to ensure the user only selects rock paper or scissors.
    while True:
    #The User input will be lowercased to ensure it matches the varia
    #Any other input is caught and the user is notified. to try again

        #This print line is used to visually identify each new game t
        print("**********************************************************
        playerSel = input("Type rock, paper, or scissors: ").lower()
        if playerSel in ["rock", "paper", "scissors"]:
            #User is notified of their input.
            print("_____")
            print(f"{playername} has selected \n{playerSel}.")
            print("_____")
            #This line of code acts as a buffer before the next funct
            input("It's the computer's turn! (Press enter)")
            #computerSel variable does not get calculated here. The f
            computerSel = computerturn()
            #The rpsgame function is called to compare the playerSel
            rpsgame(playerSel, computerSel)
            #This line of code will save the user input to be the new
            #reused in other functions. This return value MUST be las
            return playerSel
        #The else is any other possible option or typo entered by the
        else:
            print("Whoops, try again!")
            introduction()
```

If a user inputs a string from the list in the **if** statement, it continues to the next line of code, **computerSel=computerturn()**. In simple context, the resulting output of **computerturn()** will be assigned to the **computerSel** variable (Fig. 2). This computerturn function picks a random string value of rock, paper, or scissors by the computer.

The function, **rpsgame**, is called to compare values of the player and computer. **playerSel** and **computerturn** are in the argument so the function knows to bring these two values to the next function (Fig. 3).

Note that return playerSel is used at the bottom of this condition to keep the playerSel value saved for the next function.

The other **if** condition for **else** in Fig. 1 is used to prevent the user from entering a random input that cant be used. In the event that this condition is not stated, the program will not know what to do and create an error if the user typed 'rck' instead of 'rock'.

Fig 2:

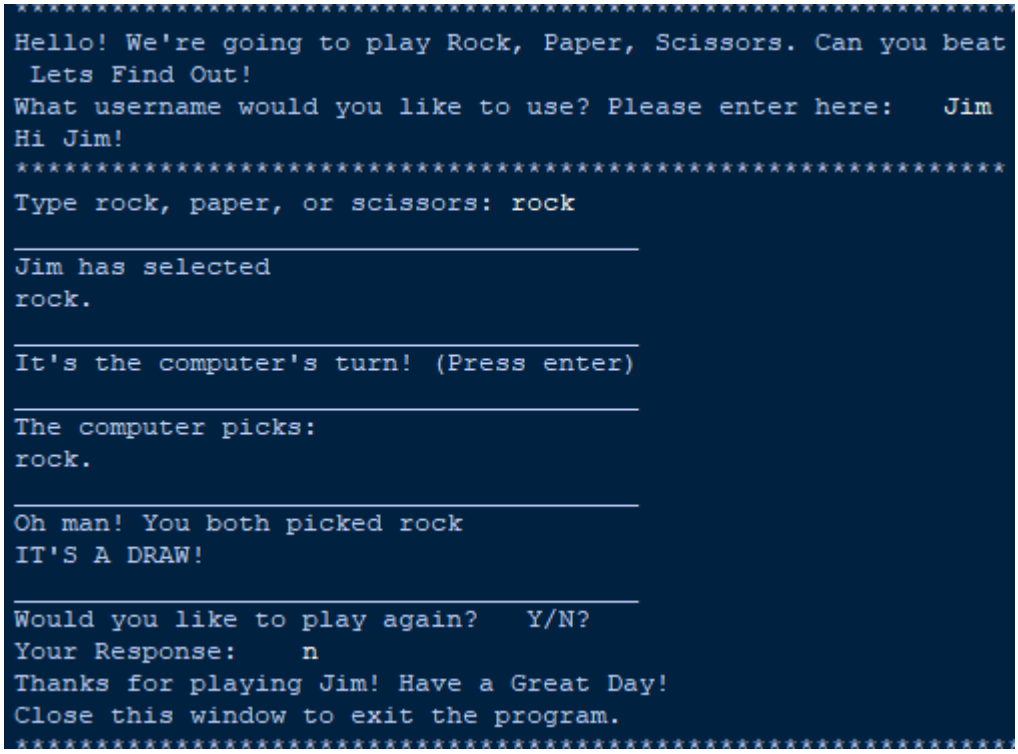
```
def computerturn():
    #using 1 and 3 as arguments, a random int
    computerSel = random.randint(1, 3)
    #A value equal to 1 is rock.
    if computerSel == 1:
        return "rock"
    #avalue equal to 2 is paper.
    elif computerSel == 2:
        return "paper"
    #The remaining value of 3 is scissors.
    else:
        return "scissors"
    #The value is returned to be used in the
    return computerSel
```

Fig 3:

```
def rpsgame(playerSel, computerSel):
    #The User is shown the computer's selection from the computerturn function
    print("_____")
    print(f"The computer picks: \n{computerSel}.")
    print("_____")
    #This IF loop will determine the game. The first if conditions defines equal values of strings as a draw.
    if playerSel == computerSel:
        print(f"Oh man! You both picked {playerSel}\nIT'S A DRAW!")
        #restart function is called.
        restart()
    #This IF condition lists all possibilities that make a player beat the computer: R > S | P > R | S > P
    elif (playerSel == "rock" and computerSel == "scissors") or (playerSel == "paper" and computerSel == "rock") o
        print(f"{playerSel} beats {computerSel}! \n{playername} WINS!")
        #Restart function is called.
        restart()
    #All other conditions are computer wins.
    else:
        print(f"{computerSel} beats {playerSel}! \nTHE COMPUTER WINS!")
```

(Output on next page.)

Output:



This program uses several **functions** to complete a game of rock, paper, scissors. With several possibilities of how any one game can turn out, an **algorithm** is required to get to the end result. The purpose for this is to ensure the program is robust, accurate, and eliminates any possibility of an error occurring from within the parameters of the code.

Misc

Modules

a.) Modules are libraries. This means that a module contains information that doesn't already exist in a code and requires that a user import the needed methods or information in the library. An example of this is using the using the random module to produce a random number.

If a section of code requires a randomly generated number, the random module needs to be imported at the beginning of the hierarchy or at the start of code (before the method is called). Using '**import random**' will import the random module/library into the program. After this, any code that pertains to this module can be used.

Example:

```
import random
```

In the case of using random, a **variable** called **computerSel** is created and assigned a method called **random.randint**. '**randint**' is short for random integer. The **argument** in this method is **1** and **3**, which will be the range of numbers that will be used for choosing a number.

Example:

```
computerSel = random.randint(1, 3)
```

Because the value of this variable is not defined with a value before executing the program, the value will always be randomly generated when the program runs this code. This type of method for using randomly generated values can be used for making games where the user doesn't know what the computer's choice will be.

There are far too many modules to list in this document and not all modules have the same purpose.

Labs/Examples

L13

Code:

```
'''
Author: Jaime Torres
Date: 3/11/2025
CS1030 Spring
'''


# A variable is a piece of data/value such as string, integar, double, or boolean
# saved on the local device's resources(memory).
# The variable acts as a box with an assigned value.

# A method is a function. Unlike a variable, a method can pass information (arguments).
# Some methods return information.

# Below are the variables that will store data.

# Lists are used to store multiple items in a single variable. When adding several items to a list, a user will need to
# add a line of syntax such as: [variable_name].append , in order to store the names. The user may then either print all
# variable items in a list or individuallly print them using a [#] with a specific number that correlates to the variable
# in the saved slot.



#Below are the variables that will be input by the user.
personname = input("Please type and enter your name:")

personbirthyear = int(input("Please enter your year of birth:"))

currentyear = int(input("Enter the current year:"))

favhobby = input("Enter your favorite hobby:")

#This creates a calculation of the person's age.
agecalc = (currentyear - personbirthyear)

#This function presents an output based on the variables and methods previously entered.
def profile(personname, personbirthyear, currentyear):
        print(f" Hi {personname}! You were born on {personbirthyear} and the year is {currentyear}.")
        print(f" This means that you are {agecalc} years old!")
        print(f" Your favorite hobby is {favhobby}!")

#This calls the function previously described.
profile(personname, personbirthyear, currentyear)
```

Output:

```
Please type and enter your name:Jimmy
Please enter your year of birth:1996
Enter the current year:2025
Enter your favorite hobby:Drawing
 Hi Jimmy! You were born on 1996 and the year is 2025.
 This means that you are 29 years old!
 Your favorite hobby is Drawing!
```

L14

Code for in-class Rock Paper Scissors Game:

```
import random

computerSel = random.randint(1,3)


playerSelection = []

1 == rock
2 == paper
3 == scissors

playerSelection = input("type rock, paper, scissors")

print("The player has selected {playerSelection}.")



def rpsgame():
    print(f"The computer picks {computerSel}")
    if playerSelection == "rock" and computerSel == "rock" or playerSelectior
        print("its a draw!")
    elif playerSelection == "rock" and computerSel == "scissors" or "paper" a
        print("Player 1 wins!")
    else:
        print("Computer wins!")

rpsgame()
```

(The draw and win conditions span too far to take a screenshot)

This code does not work as intended. If a player chooses **rock** and the computer chooses **paper**, the program still decides that the player wins. The issue was with the **elif** syntax and it was resolved later.

Output:

```
type rock, paper, scissorsrock
The player has selected rock.
The computer picks paper
Player 1 wins!
```

(Solution on next page)

The winning conditions for the **elif** section were re-typed with **parenthesis for each condition. <u>This causes the conditions to occur in the exact order they are written</u>**. Without the parenthesis, the program doesn't care about the order in which they occur, but that it only meets the conditions in any order. The program was also restructured for logical reasons and also has aesthetic string values for the outputs.

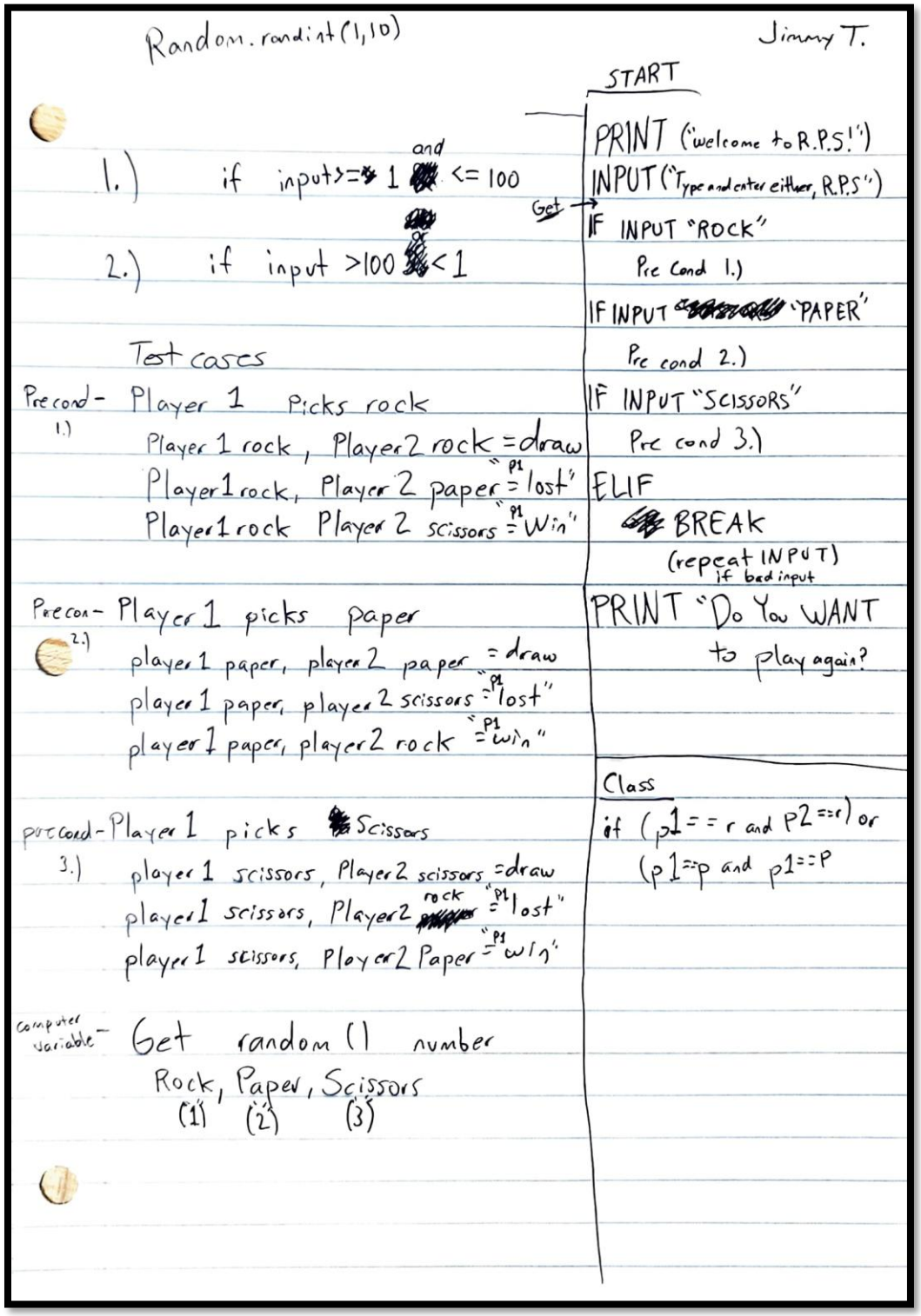
```
def rpsgame(playerSel, computerSel):
    #The User is shown the computer's selection from the computerturn function
    print("_____")
    print(f"The computer picks: \n{computerSel}.")
    print("_____")
    #This IF loop will determine the game. The first if conditions defines equal values of strings as a draw.
    if playerSel == computerSel:
        print(f"Oh man! You both picked {playerSel}\nIT'S A DRAW!")
        #restart function is called.
        restart()
    #This IF condition lists all possibilities that make a player beat the computer: R > S | P > R | S > P
    elif (playerSel == "rock" and computerSel == "scissors") or (playerSel == "paper" and computerSel == "rock") or (playerSel == "scissors" and
        print(f"{playerSel} beats {computerSel}! \n{playername} WINS!")
        #Restart function is called.
        restart()
    #All other conditions are computer wins.
    else:
```

Output:

```
Hello! We're going to play Rock, Paper, Scissors. Can you beat the Computer?
 Lets Find Out!
What username would you like to use? Please enter here:   jim
Hi jim!
**********************************************************************
Type rock, paper, or scissors: rock
_____
jim has selected
rock.
_____
It's the computer's turn! (Press enter)
_____
The computer picks:
paper.
_____
paper beats rock!
THE COMPUTER WINS!
_____
Would you like to play again?   Y/N?
Your Response:
```

L15

Pseudocode In-class work sheet:



Random.randint(1,10)

Jimmy T.

1.) if input >= 1 and <= 100

2.) if input > 100 or < 1

Test cases

Precond- 1.) Player 1 Picks rock
- Player 1 rock, Player 2 rock = draw
- Player 1 rock, Player 2 paper = "P1 lost"
- Player 1 rock Player 2 scissors = "P1 Win"

Precon- 2.) Player 1 picks paper
- player 1 paper, player 2 paper = draw
- player 1 paper, player 2 scissors = "P1 lost"
- player 1 paper, player 2 rock = "P1 win"

precond- 3.) Player 1 picks Scissors
- player 1 scissors, Player 2 scissors = draw
- player 1 scissors, Player 2 rock = "P1 lost"
- player 1 scissors, Player 2 Paper = "P1 win"

computer variable- Get random (1 number
Rock, Paper, Scissors
(1) (2) (3)

START

PRINT ("welcome to R.P.S!")

INPUT ("Type and enter either, R.P.S")

Get →

IF INPUT "ROCK"

Pre Cond 1.)

IF INPUT "PAPER"

Pre cond 2.)

IF INPUT "SCISSORS"

Pre cond 3.)

ELIF

BREAK

(repeat INPUT) if bad input

PRINT "Do You WANT to play again?

Class

if (p1 == r and p2 == r) or (p1 == p and p1 == P

L16

Code:

```
'''
Author: Jimmy Torres
Date: 4/1/2025
Program: Guess number game

Note: This game will prompt the user to guess the correct number generated
by the computer until they get it right.
'''
#This imports the random module/library
import random

#This variable is calculated using the random module. Randint will pull a
# random integar between 0 and 100 (argument).
number = random.randint(0,100)

# This print method is the first line of information presented to the user.
print("Guess the magic number between 0 and 100")

#The value of -1 is just a placeholder until values in the range is added.
guess = -1

# This while loop will tell the user if their number is too low, high or the actual number.
# It will repeat until the conditions
# make the user have an equal number.
while guess != number:
    #User input
    guess=int(input("\nEnter your guess:"))
    #Equal condition
    if guess == number:
        print(f"Yes, the number is {number}!!!")
    #guess is too high condition
    elif guess>number:
        print("Your guess is too high")
    #guess is too low condition
    else:
        print("Your guess is too low!")

#END OF PROGRAM
```

Output:

```
Guess the magic number between 0 and 100

Enter your guess:50
Your guess is too low!

Enter your guess:60
Your guess is too low!

Enter your guess:70
Your guess is too low!

Enter your guess:80
Your guess is too low!

Enter your guess:90
Your guess is too high

Enter your guess:85
Your guess is too low!

Enter your guess:87
Your guess is too high

Enter your guess:86
Yes, the number is 86!!!
```

L17

Code for multiplication table of 3:

```
'''
Author: Jimmy Torres
Date: 4/3/2025
CS1030
Program: For Loops Practice

Purpose: In Class work for understanding for loops.
'''
#The following range from 1 to 8 will print. this range is called "i".

#This is for the desired amount of multiplication problems the user wants to see.
user_input = int(input("Please enter the highest number for multiplacation table for the number three"))


#This for loop will print the range based on the user input. Because of the way Pyhthon
#is designed, a +1 value is added to the range to match the user's desired input.
for i in range (1,user_input +1):
    #multi_table variable is added to capture each time the calculation is multiplied.
    multi_table = 3*(i)
    #Every line is calculated until the range is met.
    print(f"3*{i} = {multi_table}")
#End of Program
```

Output:

```
Please enter the highest number for multiplacation table for the number three10
3*1 = 3
3*2 = 6
3*3 = 9
3*4 = 12
3*5 = 15
3*6 = 18
3*7 = 21
3*8 = 24
3*9 = 27
3*10 = 30
```

Code for printing specific items in a list:

```
'''
Author: Jimmy Torres
Date: 4/3/2025
CS1030
Program: For Loops Practice

Purpose: In Class work for understanding for loops.
'''

#This stores a list
userinputs = []

#This function saves inputs to the userinputs list.
def input_function():
    #While true starts a loop while boolean is implemented.
    while True:
        #This takes inputs by the user.
        userinput = input("Please enter an input to save, type 'done' when you are finished:\n")
        #This stops the loop.
        if userinput == "done":
            break
        #This adds all userinput values that are not 'done' to the userinputs list.
        else:
            userinputs.append(userinput)
#This starts the function when program is started.
input_function()

#This prints the item sitting in the first position which is '0'.
print (userinputs[0])
```

Output:

```
Please enter an input to save, type 'done' when you are finished:
toy 0
Please enter an input to save, type 'done' when you are finished:
ball 1
Please enter an input to save, type 'done' when you are finished:
cat 2
Please enter an input to save, type 'done' when you are finished:
done
toy 0
```