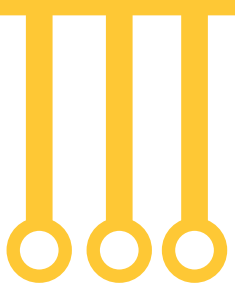
Course Book

for Programming Fundamentals



```
filename = "pfunmanual.txt"
file = open(filename, "r")
```

CREDITS:

Made by Sarwan Shah - Version 1.0
Electrical Engineering,
Class of 2021,
Habib University, Karachi.

Cover Designed by Luluwa Lokhandwala
Communication and Design,
Class of 2021,
Habib University, Karachi.

PUBLISHED BY HABIB UNIVERSITY

BOOK-WEBSITE.COM

First printing, July 2018

Programming is an Art

Contents

Getting Started

Python

1. Getting Started

1.1 What is Programming?

Programming at its core is the process of giving a computer a set of instructions, often known as an algorithm, based on which the computer performs certain calculations which can be as simple as $1+1=2$.

Modern computers can perform such simple calculations extremely fast: up to more than a billion calculations per second. Through programming, we present the computer with tasks and problems that are far more complex, but built and represented through the same basic logic or instructional blocks that are easily understandable by the computer.

The most basic form of instruction a computer can understand is binary, a stream of 1's and 0's, also known as Machine Code. However, no one codes in machine code today, instead there are pre-defined logic blocks built upon Machine Code that can be used to instruct the computer through what is known as a Assembly or Low Level Programming Language. This is still very elementary, and complex tasks and problems can be very difficult to instruct.

High level programming languages build upon this, and hence, allow easier control over the ability to instruct the computer with complex tasks and problems.

The added strain on computers is to convert these high level instructions back to Machine Code before processing it for execution. However, with the amount of computational power modern computers have today, it is a cost that can be afforded.

1.2 What is Python?

Python is an interpreted language with easy to understand semantics and syntax. This makes it a very suitable language for beginners who intend to learn programming.

Being an interpreted language means that when a Python program is executed, it executes line by line. If at any stage of the program, an error is encountered, the program's execution is halted and the error is reported in the execution shell. This allows for easy debugging.

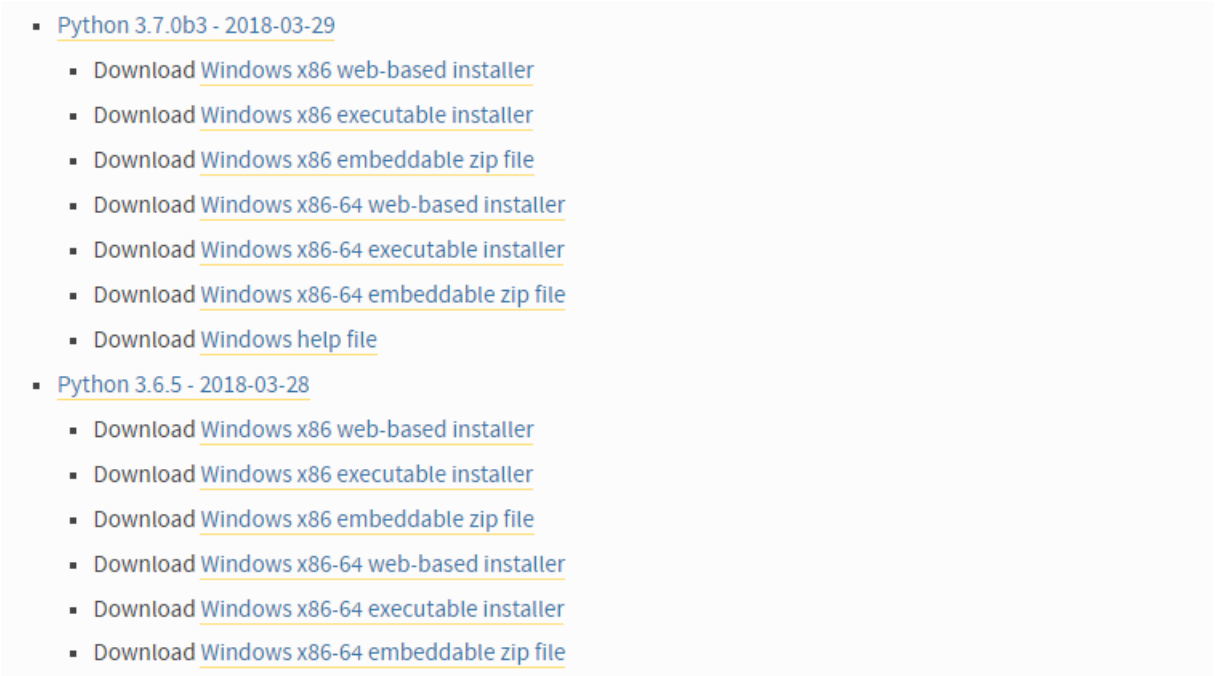
Furthermore, it is very widely used and has a large library that has pre-made programs and functions by people that can be used to perform a large variety tasks without having to code yourself.

1.3 Installing Python

To install Python first you need to know whether you are using a 32 bits or 64 bits operating system (OS). Most new computers are likely to be operating on 64 bits. If you are on Windows, then follow these instructions on this website, and if on a Mac OS then the ones on this website to figure it out. For other operating systems, feel free to Google.

Once you have determined this, proceed to this website to download the latest version of Python 3.x.x for Windows, or to this website for Mac OS.

i.e. For Windows 10 64-bit I would download 'Windows x86-64 executable installer' under 'Python 3.6.5' since it is the latest stable release: which is evident from the fact that it has no alphabetical extension to its name as in other versions.



Once the download has completed, double click the installer to initiate the installation process. Proceed with a default installation and be sure to tick the "Add Python 3.x to PATH" box while installing.

After the installation has completed, to verify, search and run the Command Prompt (cmd) on Windows. Type 'python' into the Command Prompt shell. You should get a messaging verifying its presence, or else, alternatively you could search for the presence of the application 'IDLE' on your system.

If you get an apparent error, feel free to Google it or visit here to under see the installation process again.

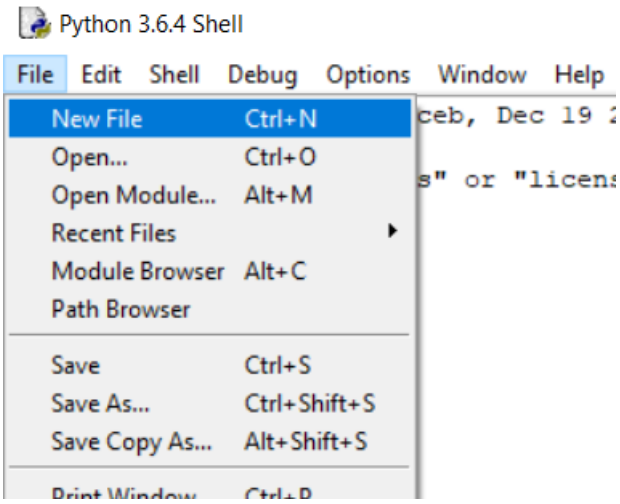
1.4 Using IDLE

Now obviously since programming requires us to type code as instructions, we need an environment where we can type this code as text. We cannot do this in text editors like Microsoft Word or Word-pad. Instead, we have more specialized text editors and IDE's (Integrated Development Environment) for this purpose like Sublime Text, Atom, Anaconda etc.

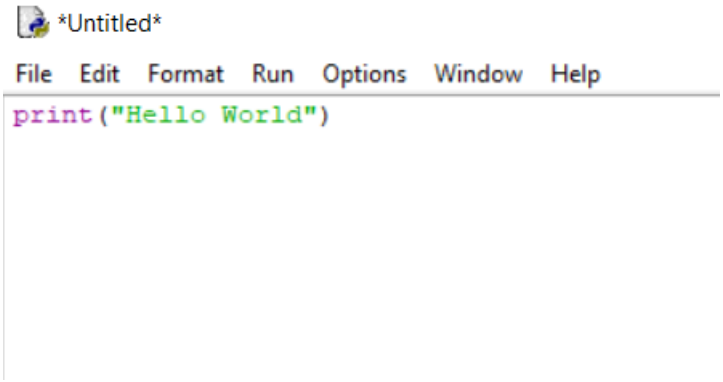
The advantage of these specialized editors or IDE's is that they allow more clarity, flexibility and tools such as syntax-coloring, debuggers etc that aid in the process of coding.

However, for now we will stick to the basic built-in IDE (Integrated Development Environment) provided with Python itself, which is known as IDLE. To access IDLE, in the run or application search bar of your operating system type 'IDLE'.

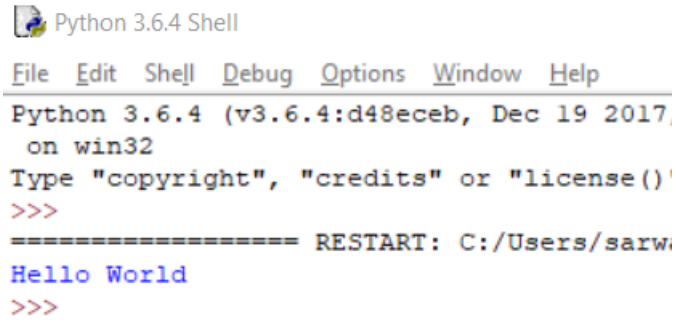
An application named 'IDLE (Python 3.x 32/64 bit)' should pop up if you installed Python correctly. Run this application. A Python Shell should pop open:



Go to File > New File and click it. This should open a new window in which you can type text. This is your IDLE text editor, where you will be typing your program code. For your first program type in 'print("Hello World")' as such:

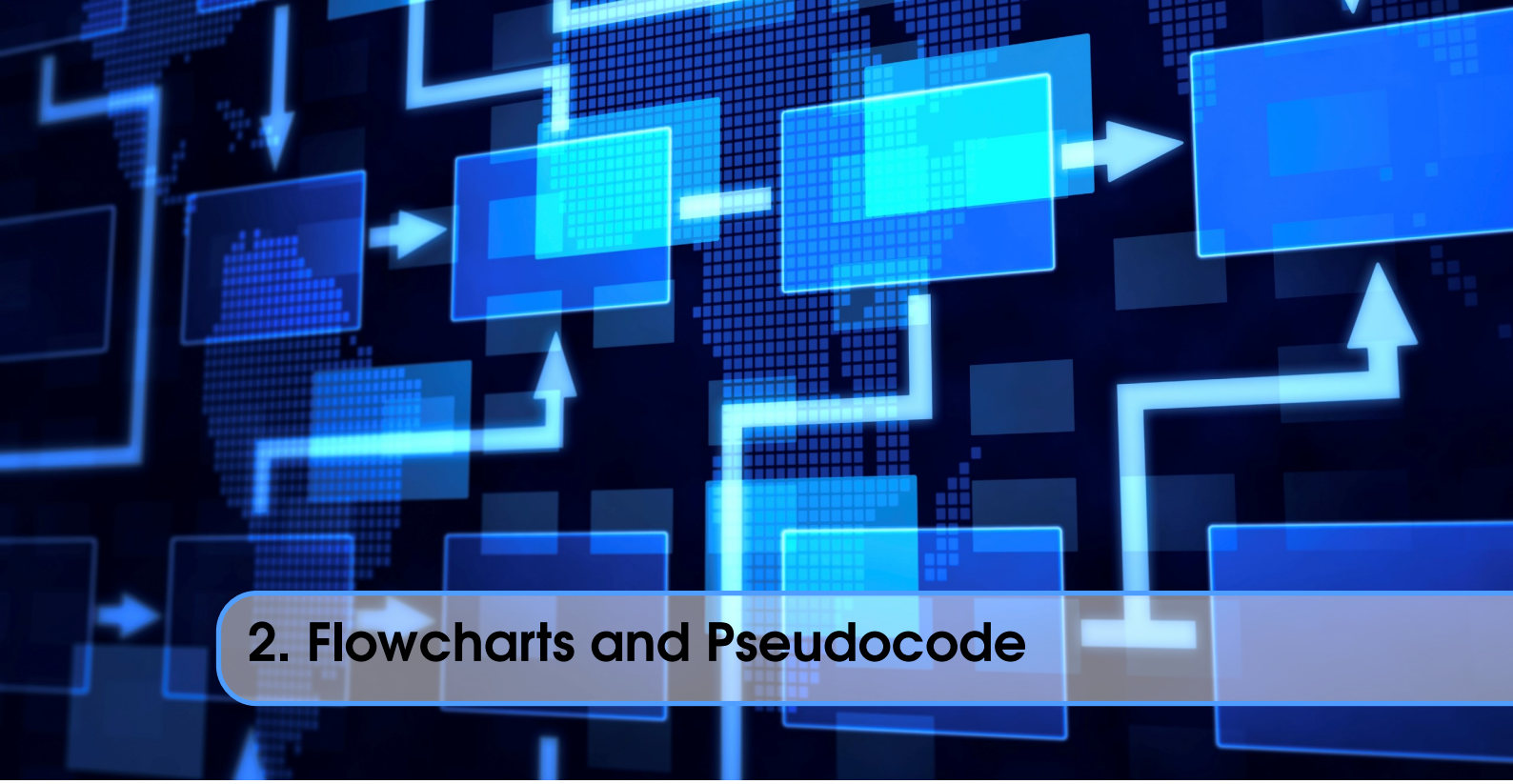


Now go to File > Save and save it as 'anyname.py'. Now go to Run > Run Module. This should execute your code, and if you have followed along correctly so far then this should show up in the Python Shell:



The results of your executed code or any errors are reported through the Python Shell. Alright! So this sets us up straight for using Python to code in the future labs. For any more details on IDLE's usage or other IDE's, if you are curious, Google is your friend!

2 Flowcharts and Pseudocode



2. Flowcharts and Pseudocode

2.1 What is an Algorithm?

Before we start understanding what flowcharts and pseudo-codes are, and how they related to programming fundamentals, we must first understand what an algorithm is.

An algorithm is a set of steps or instructions to accomplish a task. For example, you could have a set of instructions laid out to log onto facebook, ride a bike, or make a grilled cheese sandwich. Similarly, in Computer Science, an algorithm takes in some input data, does some calculations on it or using it, and outputs a result.

Algorithms lay at the very foundation of Computer Science and are an integral part of almost anything that is digital, from tasks as simple as browsing the web, to computing and predicting climate trends across the globe.

2.2 What is a Flowchart?

A flowchart is a diagrammatic representation of a program that runs on a certain algorithm. It helps in laying out the basic structure of a program and understanding the logic on which it works.

Flowcharts are always drawn top to bottom. Differently shaped boxes and symbols represent different states in a flowchart. These are connected using arrows, whose show the flow of control through each of these states.

Below are listed some essential flowchart symbols:

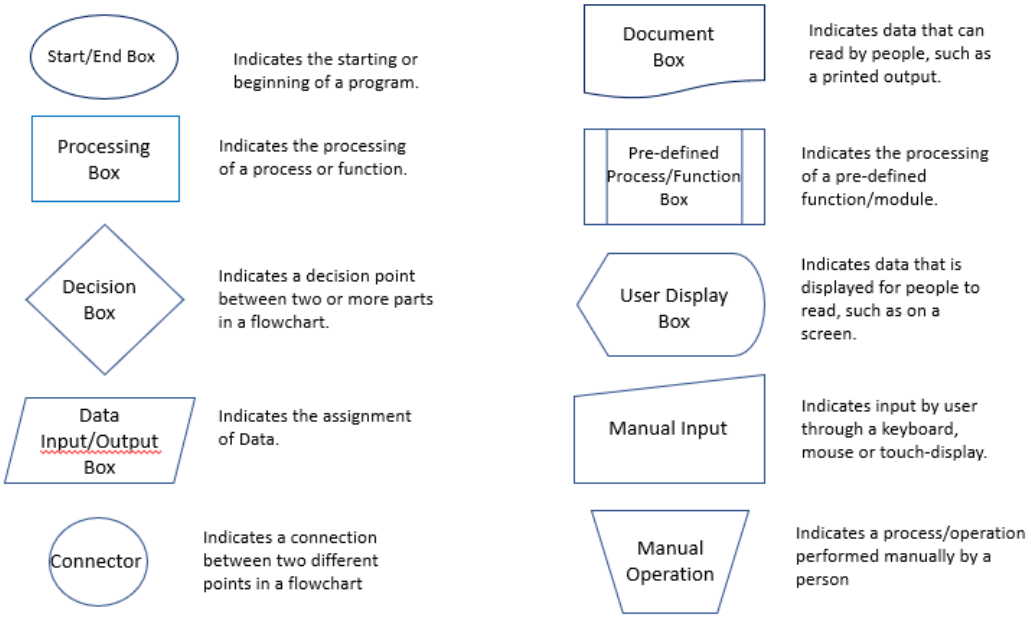


Figure 2.1: Essential Flowchart Symbols

A complete list of the standard flowchart symbols can be found here.

A simple program that takes input two numbers, x and y, and returns the average can be represented using a flowcharts as such:

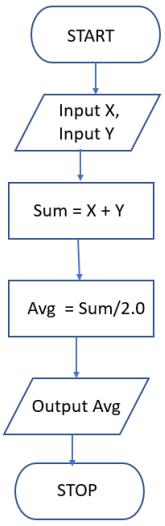


Figure 2.2: Finding Average

2.3 What is Pseudocode?

Every programming language has its specific syntax or structure. Unlike a programming language, pseudocode are simple steps that directly explain how a particular code or program functions, without giving much concern to syntax. With aid of already available pseudocodes for a programs, programmers can write programs in the language of their choice.

A pseudocode is a step by step description of a program and does not follow any syntax. It is in essence a textual representation of a program and like flowcharts, it helps in laying out the basic structure of a program and understanding the logic on which it works. This takes us one step further to actually writing a program.

The example of a pseudocode for the process of making tea would be something like this:

1. Pour water into the kettle
2. If kettle is full, stop pouring, else keep pouring
3. Next, add tea powder, milk and sugar
4. Turn on the stove
5. Place the kettle on the stove
6. If tea is ready, turn stove off, else keep it on until the tea is ready

The pseudocode for our flowchart in figure 2.2 above would be as follows:

```
Begin
 input x, y
 sum = x + y
 print sum
End
```

2.4 Lab 1 - Activity

For this activity you are required to make groups. The number of members per group will be instructed to you by your instructor.

Your task is to guide your partner to the whiteboard, grab a marker and then come back, give the marker, and sit down. You will need to approximate the number of steps your team member has to take to complete this task. Write pseudocode that will instruct your team member to complete this task.

The instruction you can give your partner are:

Get up (Up)
Move 1 step (Move)
Rotate 90 degrees clockwise (90 CW)
Rotate 90 degrees counterclockwise (90 CCW)
Grab marker (Grab)
Give marker (Give)
Sit down (Down)

2.5 Lab 1 - Exercises

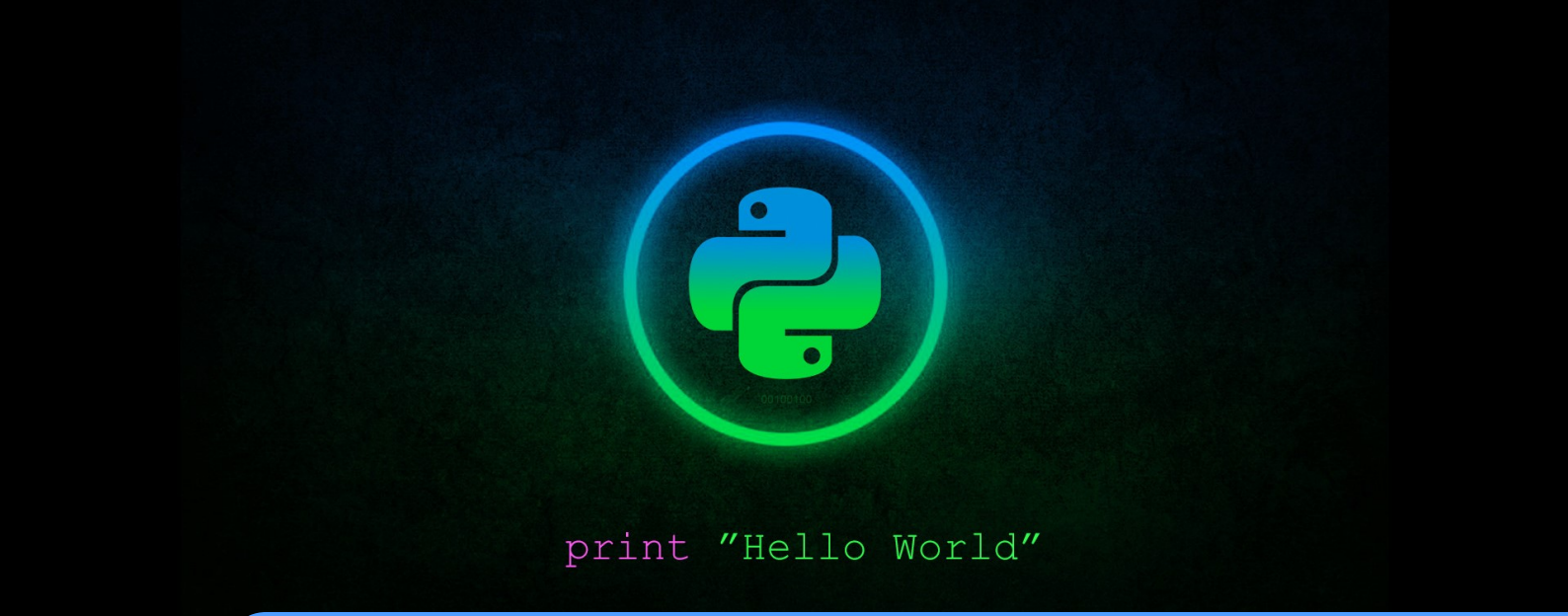
Q1.(a) Habib University uses an hourly wage system for students that would like to work during the summers as teaching or research assistants. The wage is set at Rs.150 per hour. The per week work hours limit is 40.

Your friend at another university has decided to work during the summers, but he cannot figure out how to calculate his weekly income. Write a flowchart and pseudocode for a program that may help him do this. Your inputs will be the weekly hours committed and wage, and output will be the weekly income.

Q1.(b)Your friend was very thankful for your help, however, he does have one other request. Unlike Habib University, his university does not have a limit on the hours per week, infact, it pays 1.5 times the normal wage for hours committed over 40.

Your friend has a lot of free time at his hand, and would like you to help him figure out how he could calculate his weekly income if it exceeds 40 hours. Write a flowchart and pseudocode for a program that may help him do this. Your inputs will be the weekly hours committed and wage, and output will be the weekly income.

Strings, Variables and Conditionals



3. Strings, Variables and Conditionals

3.1 Operators

The ability to perform various arithmetic and boolean (comparison) operations lays at the very foundation of programming. As such Python incorporates certain arithmetic operators and comparison operations that are innate to the language, while others are built upon these basic operators and have to be imported from libraries.

The following is a list of some of the basic arithmetic

1. '+' - addition
2. '-' - subtraction
3. '*' - multiplication
4. '/' - division
5. '//' - floor division (rounds down non-integer results to a integer)
6. '**' - exponent
7. '%' - modulus (gives the remainder of a division)

Open your IDLE python shell and try some of these out!

```
>>> 4 + 6
10
>>> 6 - 4
2
>>> 4 * 5
20
>>> 14/6
2.3333333333333335
>>> 14//6
2
>>> 2**4
16
>>> 16%4
0
```

Figure 3.1: Arithmetic Operators

For other mathematical operations such as the square root of a number or trigonometric operations we can make functions of our own based on these basic operators, which is something we'll learn later.

For now you must know that we have libraries available, that have pre-defined functions, that can do these operations for us. One such very commonly used library is called 'math'.

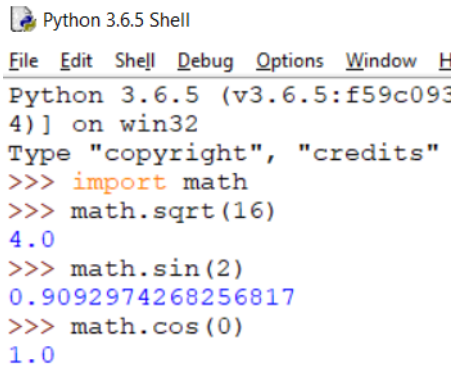


Figure 3.2: Imported functions

In figure 3.2, we use 'import' to import the 'math' library to our program. 'import' is used to import libraries in python and is one of the 33 reserved Python keywords that cannot be used for any other purpose.

Another very important set of operators are the comparison operators. These perform comparisons between data and output boolean (True or False) results. They include the following:

1. '==' - equal
2. '! =' - not equal
3. '>' - greater than
4. '<' - less than
5. '>=' - greater than or equal to
6. '<=' - less than or equal to

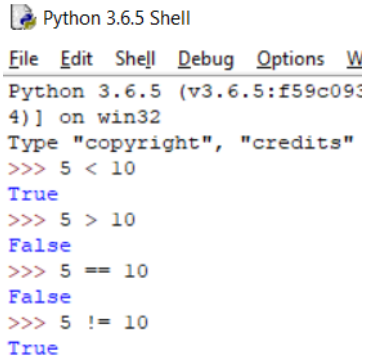


Figure 3.3: Comparison Operators

Other operators that cover symbols like '&', '|' etc include Logic and Bitwise Operators, these operate on numbers when they are represented as a binary sequence of 1's and 0's.

3.2 Data Types

The data stored by a program in memory can be of many types. For example, a persons bank balance must be stored as a numeric value, and his name as a stream of characters i.e. as a string. For this purpose we have different data types that are stored and usable in a manner best suited to the purpose they serve.

The following is a list of some basic data types:
1. Integers - int - is a numeric data type for storage and operations on integer numbers.
2. Floating Point - float - is a numeric data type for storage of decimal numbers.
3. Complex - complex - is a numeric data type for storage of complex numbers.
4. String - str - is a character based data type used for storage of a stream of characters.
5. Boolean - bool - is a logic based data type used for storing truth values: True or False.

Other data types include structures like lists, dictionaries, tuples etc, which we will learn later in the course.

A string representing a number can be easily converted to a integer or float using the operation int(*insert expression*) or float(*insert expression*) and vice versa using str(*insert expression*).

Similar conversions can be made between other data types too, on how these work and behave can be figured out by self exploration, some are demonstrated below.

```
>>> type(2)
<class 'int'>
>>> type(2.05)
<class 'float'>
>>> type(2+3j)
<class 'complex'>
>>> type('Is this a string?')
<class 'str'>
>>> type(True)
<class 'bool'>
>>> int('101')
101
>>> str(101)
'101'
>>> bool(1)
True
>>> bool(0)
False
>>> int(True)
1
>>> int(False)
0
```

Figure 3.4: Data Types Demonstration

3.3 The Print Statement

A very important aspect of every programming language is the ability to output/display data to the user via the console, which may be results, instructions or to help with debugging your program by outputting data at different steps to see where the problem occurs. For this purpose, in Python, we have what are called 'print' statements. They can be used over any data.

No data from the program displays out in the console unless a print statement is used on it.

Below is a demonstration of how these work:
Input:

```
print("Hello World")
print("This is your first program")
print("Oh btw, a stream of characters")
print("in inverted double or single commas")
print('is known as a \'string\'.')

# This is a comment, you don't need to neccesarily
#include this in your program.
print('')
print('A string is a data type.')
print('')
# ------------------ INTERESTING -----------------
print("You could also print other data types like integers\n")
print(5)
print('\n')
print("Or even the result of an arthmetic\
\nOperation between two integers")
print(25*4)
print("Answer:", 100)
print("a"*5)
```

Figure 3.5: Input Code

Output:

```
Hello World
This is your first program
Oh btw, a stream of characters
in inverted double or single commas
is known as a 'string'.

A string is a data type.

You could also print other data types like integers

5


Or even the result of an arthmetic
Operation between two integers
100
Answer: 100
aaaaa
```

Figure 3.6: Output in shell

It is recommended that you take out a moment and try this out yourself in IDLE. Some of the stuff might not be understandable, see if you can figure it out or ask your instructor.

3.4 Variables

So far we have only been performing operations on directly input data, with no storing of any data or data input from the user. Now we will look at how data is stored and input from the user.

For the purpose of storing any data we use variables. Variables are assigned data using an assignment operator '=' and any Python data type can be assigned to a variable. When this variable is later called or mentioned somewhere in the program, it links back to the data it was assigned.

Variable names are case-sensitive, and can only start with a letter or underscore, after which numbers may also follow. It is essential that you give sensible names to your variables, so that your code is easily understandable when you or someone else refers to it.

Variables can also be re-assigned with new or updated data at a later step in the program. A variable may also be assigned to another variable. You can also assign multiple data to multiple variables, and input data from the user and assign to a variable

The following snippet of code demonstrates the use of variables, try it out:

```
import math

Radius = 3
_qt_pi = math.pi
area_0f_circle = _qt_pi*(Radius**2)
print("The area of a circle of radius", Radius, "is", area_0f_circle)

Radius = 6
area_0f_circle = _qt_pi*(Radius**2)
print("The area of a circle of radius", Radius, "is", area_0f_circle)

Width, Height = 10, 5
print("The area of dimensions", Width, "x", Height, "is:", Width*Height)

name = input("What is your name: ")
rect_width = int(input("Rectangle Width: "))
rect_height = int(input("Rectangle Height: "))
rect_area = rect_width*rect_height
print("Okay,", name,"the size of your rectangle is", rect_area)
```

Figure 3.7: Variables

It is also important to note that data assigned to a variable needs to consistent/valid with the operations that are performed on it to avoid errors.

3.5 Intro to Strings

Strings are a data type in Python. They are represented as a stream of characters between inverted single or double commas.

Strings are a immutable data type, which means they cannot be changed once assigned to variable. However, the original string can be merged and extended with other strings using what is known as concatenation, and then assigned to the variable as an altogether new string.

In other words, you can't change the original string, but you can replace it with a new updated one. For example,

```
name = "Sarwaan"
# There is no method or way to change
# or remove the extra 'a', while maintaining
# the original string. However, you can
# re-assign 'name' to a updated new string

name = "Sarwan" # Re-assigning
name = name + "Shah" # Concatenating with original string and re-assigning
print(name) # Output: SarwanShah
```

Figure 3.8: Strings are immutable

There is much more to play around with strings that we'll look into a later section.

3.6 Conditionals

So far we've been looking at programs that are linear in nature, such that, after every line of code a executes one after another. These are called straight-line programs. There is not a lot interesting that can be achieved by such programs, infact they are quite boring.

The ability to add conditions, based on which a different set of code is executed makes programs far more interesting and extends the scope of functionality they can achieve. For this purpose we have what are known as conditionals.

Conditionals allow us to branch programs, based on the result of a condition that may be true or false. There are four key features of a conditional:

1. An expression/condition that evaluates true or false (if)
2. A block of code that executes if true
3. An optional expressions/conditions that evaluates if the previous one was false (elif)
4. An optional block of code that evaluates if all above is false. (else)

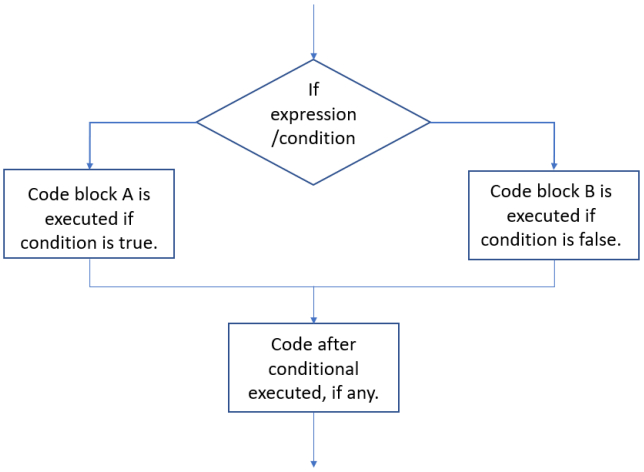


Figure 3.9: Conditional Flowchart

The general code structure of a conditional block is as follows:

```
if expression/condition:
      block of code
elif expression/condition:
      block of code
else:
      block of code
```

The elif and else statements are optional. If not included, the code will move to the next statement in the program after the conditional. Furthermore, even indentation of the code (spacing) under conditional statements helps Python know the code is a part of the conditional.

The following program demonstrates how conditionals are used, it is recommend that you type and test this program out:

```
username = "Sarwan"
password = "2021"
balance = 420000

# Asking for user input
input_username = input("Username: ")
input_password = input("Password: ")


# Conditionals what to excute based on deciding based on user input
if input_username == username and input_password == password:
    print('----------------------------------------')
    print("User logged in as", username)
    print("Enter 1 to check balance")
    print("Enter 2 to change password")
    print("Enter 3 to log out")
    print('')
    user_input = input("Enter: ")
    print('----------------------------------------')

    if user_input == '1':
        print(balance)
        print("Done, exiting program.. ")

    elif user_input == '2':
        input_password = input("Enter new password: ")
        password = input_password
        print("Done, exiting program.. ")

    elif user_input == '3':
        print("Logging out, exiting program.. ")

    else:
        print("Incorrect choice, exiting program..")

else:
    print("Wrong username or password, exiting program..")
```

Figure 3.10: Conditionals

3.7 Lab 2 - Exercises

Q1. Temperature is measured in 3 scales: Kelvin, Celsius and Fahrenheit. The derived SI Unit for temperature is Celsius, and is used globally to represent temperatures, except in the United States, where Fahrenheit is used.

You work for a weather forecasting company that operates globally. You need to write a program that takes input from the user the temperature in Fahrenheits as an integer, and converts it to temperature in Celsius for international broadcasting purposes.

The mathematical formula for converting Celsius to Fahrenheit is $F = 1.8 * C + 32$

- - - - - SAMPLE- - - - -
Input: 25

```
30 degrees Celcius is equal to 86.0 degrees Fahrenheit.
```

Input: 30

```
35 degrees Celcius is equal to 95.0 degrees Fahrenheit.
```

Q2. Write a program that calculates the roots of a quadratic equation. The program takes input 3 integers that assigned to the variables a, b, and c that correspond to the coefficients of a quadratic equation.

Using the determinant formula, D = b**2 - 4ac, first determine the nature of the roots and print out a statement accordingly

1. if D > 0 (This equation has two real and unique roots)
2. if D < 0 (This equation has no real roots)
3. if D = 0 (This equation has two equal roots)

Next, if the equation has roots, then calculate them using the quadratic formula:
Smaller root = (-b - sqrt(D))/2a
Bigger root = (-b + sqrt(D))/2a

Hint: You are required to import the math library for taking the square root of the determinant. When printing the equation you will need to set the separator (by default it is set to a single space) to empty i.e. sep = ”, to print out the equation as required for removing spaces.

For example, print("My name is", name, sep = ”). The comma is the separator and is replaced by an empty space now when the statement executes. Feel free to further Google how the separator works.

Alternatively, you could use string concatenation to output the equation in the format given in the sample on the next page. Look up what string concatenation is.

- - - - - SAMPLE- - - - -
a = 1, b = 5, c = 6

```
a: 1
b: 5
c: 6
Equation:1x^2 + 5x + 6 = 0
This equation has two real and unique roots
Roots: -3.0 -2.0
```

a = 1, b = -2, c = 1

```
a: 1
b: -2
c: 1
Equation:1x^2 + -2x + 1 = 0
This equation has two real equal roots
Roots: 1.0 1.0
```

Q3.(a) Write a program that takes input a number as a string of length 4 from the user, and breaks it down into thousands, hundreds, tens and units.

- - - - - SAMPLE- - - - -
input: 1123

```
Number: 1123
Thousands: 1
Hundred: 1
Tens: 2
Units: 3
```

Hint: You'll need to use the int() function, and maybe the str() function too. Look up their working.

Q3.(b) Write a program that separately takes input from the user the thousands, hundreds, tens and units of a number as integers, and builds it up to the original number that is printed out as a string.

- - - - - SAMPLE- - - - -
input: 1 1 2 3

```
Thousands: 10
Hundreds: 22
Tens: 15
Units: 15
12365
```

Hint: You'll need to use the str() function.

IV

Loops

THINK TWICE
CODE ONCE!

4. Loops

In the previous section we learned about conditionals, and saw how greatly they increase the scope of functionality that can be achieved. Similarly, iteration using loops also vastly increases the scope of functionality that can be achieved using a program. Iteration is the act of repeating a process.

4.1 While Loops

In Python, 'while' loops are used to continuously repeat over a block of code. The looping ends if the expression/condition given with the while loop becomes false, or if the block of code reaches a 'break' statement. After this, the next line outside the loop executes.

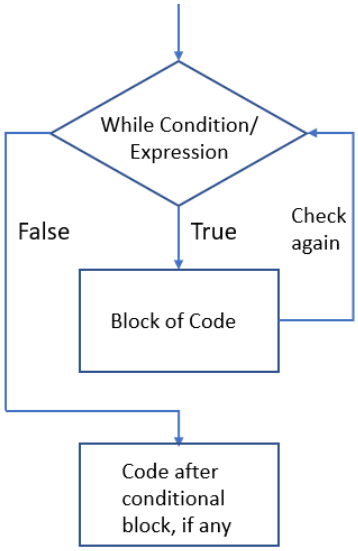


Figure 4.1: While loop flowchart

The general structure of a while loop is as such:

```
while expression/condition:
      block of code
```

For example, trying the following code will repeatedly print "This will never stop" in the Python shell unless you interrupt the shell using Ctrl + C. Try it out:

```
while True:
    print("This will never stop")
```

Figure 4.2: while True

Code running over and over again infinitely without changing anything does not really help much, what we want is that the loop changes something over time until it reaches a point where our 'while' expression is no longer true.

This is usually achieved by incrementing or changing the data assigned to a variable (that is also a part of the while expression) in the block of code below the loop. The following snippet of code demonstrates this, by printing numbers 1 to 10 on separate lines, try it out:

```
i = 1
while i <= 10: # Repeat the code while i is less than or equal to 10.
    print(i)
    i = i + 1 # This increase the value of i by 1 everytime the loop repeats.
              # This could also be written as i += 1.
```

Figure 4.3: while with a condition

Another way to stop a while loop is to include a 'break' statement under a conditional in it. This will break the loop at that point, and execute the next available line outside the loop. The following code behaves the same as the code in figure 4.3:

```
i = 1
while True:
    print(i)
    if i == 10:
        break # This is Python keyword that breaks loops.
    else:
       i += 1 # Same as i = i + 1
```

Figure 4.4: while with breaking condition

On the contrary, there also exists a 'continue' statement in Python that force repeats the code under a loop.

4.2 For Loops

'for' loops offer more or less the same functionality that can be achieved using a 'while' loop. However, they tend to be a shorter way to instruct the interpreter. In fact, the interpreter actually converts a for loop to a while loop in the background before executing it.

The general structure for executing a range based 'for' loop is:

```
for somevariable in range(x, y, z):
    block of code
```

Here, x, y and z are called parameters of the range function. The range function makes are

The x is the starting integer value of 'somevariable' that you'll name, like the $i = 1$ in figure 4.3. By default this is 0

The y defines the limit integer value to which this variable will be incremented/decremented, not inclusive of y. Like the boolean expression $i <= 10$ in figure 4.3. This always needs to be specified, and has no default value.

The z defines the integer value by which the variable increments or decrements at each step, like the i += 1 in figure 4.3. By default this value is 1. The number of steps taken to reach the limit integer value is also the number of times the block of code under the loop executes.

The following code demonstrates the working of range based 'for' loops:

```
for i in range(11): # (x = 0, y = 11, z = 1), loop runs from 0 till 10.
    print(i)        # Hence, it repeats 11 times.

print('')

for i in range(1, 11): # (x = 1, y = 11, z = 1), loop runs from 1 till 10.
    print(i)           # Hence, it repeats 10 times.

print('')

for i in range(0, 11, 2): # (x = 0, y = 11, z = 2), loop runs only even numbers.
    print(i)              # Hence, it repeats 5 times.

print('')

for i in range(10, -1, -2): # (x = 10, y = -1, z = -2), loop runs from 10 till 0.
    print(i)                # Hence, it repeats 5 times too.
```

Figure 4.5: for some range

Note: In figure 4.5, the second loop behaves same as the 'while' loop in figure 4.3.

These loops are primarily helpful when indexing over data. The 'for' loop can also directly iterate over the data itself, instead of numbers that can then be used to index over data. In fact, in reality, the range function above generates data of a set of integers, as per the specification, over which the loop then iterates.

The general structure for such a 'for' loop is:

```
for somevariable in somedata:
    block of code
```

The length/size of data available is the number of times the block of code under the loop repeats. For example, suppose this data were to iterate over a string:

```
name = "Sarwan"
for character in name: # Note: 'character' is a random name. It could be
    print(character)   # anything, but it is always a good practice to
                       # give variables sensible names.
```

Figure 4.6: for over a String

This will print all the characters of the string named 'Sarwan', starting from 'S' till 'n', on separate lines each. Every time the loop runs a new character is assigned to the variable 'character' from the string.

4.3 Lab 3 - Exercises

Q1.(a) Write a program that takes input an integer 'a' from the user, counts from 0 till 'a' (inclusive) and prints every integer in between and prints their total sum.

Hint: Make use of range based 'for' loops. The user input will need to be assigned as a a parameter to the range function.

- - - - - SAMPLE- - - - -
Input: 50

```
a: 3
0
1
2
3
Total:  6
```

Input: 75

```
a: 4
0
1
2
3
4
Total:  10
```

Q1.(b) Write a flowchart for the program in part (a). Hint: Use a while loop to represent the for loop logic.

Q1.(c) Write a program for part (a) using a while loop.

Q2.(a) Write a program that takes input two integers, 'a' and 'b' from the user, and counts/iterates from 'a' to 'b' prints the total average of every integer in between (a and b inclusive). You are required to use a 'for' loop for this part.

- - - - - SAMPLE- - - - -
Input: 10 100

```
a: 10
b: 100
Average:  55.0
```

Input: -10 20

```
a: -10
b: 20
Average:  5.0
```

Q1.(b) Write pseudocode for the program in part (a). Hint: Use a while loop to represent the for loop logic.

Q2.(b) Write a program for part (a) using a while loop.

Q3. Write a program that takes input an integer 'a' and prints the multiplication table of that number.

- - - - - SAMPLE- - - - -
Input: 5

```
a: 5
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

Q4. Factorial of any number n is represented by n! where
$n! = 1*2*3*....*(n-1)*n$. i.e.
$5! = 1*2*3*4*5 = 120$
$4! = 4*3*2*1 = 24$
$3! = 3*2*1 = 6$

Write a program that takes input an integer, and calculates and prints its factorial.

Note: $1! = 1$ and $0! = 1$

- - - - - SAMPLE- - - - -
Input: 5

```
a: 5
factorial:  120
```

Q5.The Fibonacci sequence is a series of numbers where a number is found by adding up the two numbers before it. Starting with 0 and 1, the sequence goes 0, 1, 1, 2, 3, 5, 8, 13, 21, 34....

Write a program that input an integer 'n' and prints the Fibonacci sequence till the nth term. Hint: You'll need to use 'while' loops.

- - - - - SAMPLE- - - - -
Input: 13

```
n: 13
0
1
1
2
3
5
8
13
21
34
55
89
144
```

V

Functions - I

5. Functions - I

With the addition of loops to our programming skill set, we've seen we can perform powerful tasks such was finding the factorial of a number or generating the Fibonacci sequence to the nth term, where the limit is only our computational power.

However, so far our programs are one dimensional. Real programs tend to be multi-dimensional. They involve the combination of several smaller tasks, that are combined and used together, often repeatedly, to perform a larger complex task.

This is what functions helps with, they allow us to neatly structure a program as a collection of smaller tasks. A good analogy would be consider a car to be a program, then its engine, transmission system, braking system, suspension system etc are all functions that combine and work together to make what we know as a car.

5.1 What are functions?

Mathematically, a function is a process or relation that associates two sets of elements. $f(x) = 2x$ is a mathematical function. We give it in an input x value, and it gives us an output f(x) value depending on the process/relation it performs. A functions can also be multi-variable i.e. $f(x,y) = 2x+4y$

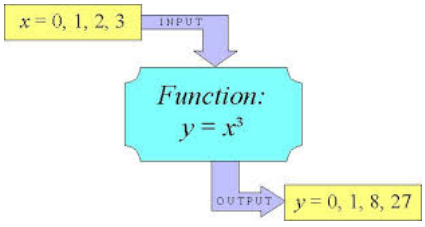


Figure 5.1: A Function

In context of programming, a function is a named sequence of instructions/code that performs some computation. Like a mathematical function, it may take some inputs/parameters (though not necessarily), often known as passing 'arguments' to the function, and use those inputs/parameters in its computation.

Once defined, these named sequence of code can then be called else where in the program when needed. Hence, making the program cleaner of redundancies and more flexible.

Functions are also called Methods and Procedures.

5.2 Defining Functions

The general outline for defining a function in Python:

```
def name_of_function(argument_1, arugment_2, could_be_any_name):
      block of code (function body)
```

Here 'def' is a python keyword used to initiate the process of defining a function. The word 'argument' in this context is synonyms with parameter or input. These inputs are actually variables, that are defined only within the scope of the function body.The function body being the block of code that executes when the function is called somewhere in the program.

Note: A function can have no inputs at all too i.e.

```
def some_function():
      block of code (function body)
```

Also, like with conditionals, even-indentation (spacing) is important to differentiate the function body code from other code.

5.3 Calling Functions

The following is an example of a function that prints the discounted price of an item that maybe made for a POS (Point of Sale) program of a supermarket.

```
def discounted_price(item_price, discount):
    new_price = item_price * (1 -  discount/100)
    print("New price after " + str(discount) + "% discount is: " + str(new_price))
```

Figure 5.2: Discount function

The function is defined by the name 'discounted_price' and takes two arguments/parameters: item_price and discount.

When the function is called somewhere the program, at that point the program skips to the function body of that function and executes the code in it.

If the function takes any input parameters/arguments then in the function call you will need to either pass values directly, or through variables that are associated with some values.

The type of the data that is passed to the function arguments/parameters needs to be consistent with how the arguments/parameters are used in the function body.

This is what a function call looks like:

```
def discounted_price(item_price, discount): # Defining function
    new_price = item_price * (1 -  discount/100)
    print("New price after " + str(discount) + "% discount is: " + str(new_price))

discounted_price(200, 20) # Function call
#--------------------- OR ---------------------------#
price = 300
dis = 30
discounted_price(price, dis) # Also a function call
```

Figure 5.3: Function Call

```
New price after 20% discount is: 160.0
New price after 30% discount is: 210.0
```

Figure 5.4: Output

The first function call directly binds 200 to 'item_price' and 20 to 'discount'. The second function call binds the variables 'price' and 'dis' to 'item_price' and 'discount' respectively, which are in turn binded with the values 300 and 30.

Similarly, instead of a variable our function call could also pass an argument as an expression i.e. discount_price(220+180, 5+5), and the expressions would be evaluated before being binded.

It is worth noting that a function can only be called in a line after it has been defined. Any attempts to call a function somewhere in the code before it has been defined will result in an error.

5.4 Variable Scopes

Let's look at another example:

```
y = 3
def sum_int(x):
    w = 10
    y = 5
    z = w + x + y
    print('w =', w,'x =', x ,'y =', y)
    print('z =', z)

sum_int(3)
print('-----------')
print('y =', y)
print('-----------')
w = 7
sum_int(3)
print('-----------')
print('w =', w)
```

Figure 5.5: Variable Scope

```
w = 10 x = 3 y = 5
z = 18
-----------
y = 3
-----------
w = 10 x = 3 y = 5
z = 18
-----------
w = 7
```

Figure 5.6: Output

Now let's see what's happening here? In the first line we bind 'y' = 3. In the next-line starts our function definition that takes 'x' as a parameter. In the function body we bind 'w' to 10 and 'y' is re-assigned to 5. The function is supposed to print the sum of the integers w, x, and y.

In the next statement we make a function call to the above defined function, with the integer 3 as the passing argument (this binds x to 3). The function prints w = 10, x = 3, y = 5, and z = 18 (which is the sum of w, x and y). This is fairly reasonable.

In the next statement it goes about to print y, which seemingly should be 5 since we re-assigned it in the function, but it prints y = 3.

This is because the variable y exists in two different scopes or name spaces: global and local. Whenever we create a function, with it we create a name space or scope that is unique or local to the functions body. It means that variables/assignments or any other assignment operations that take place inside the function are limited or local to the function only, and do not exist or lead to changes outside it i.e. in the global scope.

For this reason, in the next line when we bind 'w' to 7 and re-call the function, the value of 'w' remains 10 inside the function and prints as 7 outside.

On the other hand, if we were to remove y or w from the function and then assign them outside before calling the function, then the function would use their values assigned outside the function. This is because they exist in the global scope, which covers the function scope too unless they are separately assigned in it.

5.5 Return Statements

The functions we have written so far were void in nature. They only involved printing statements to the console. In other words, the functions we not 'returning' or yielding any data that could be used or passed as parameters to other functions or help with other computations in the global scope or parent.

This is because any data changes or generation was limited to the local scope of the function. For outputting data to the parent scope we have what are called 'return' statements in the function body.

These statements break the function at the point they are executed in the function body and output or 'return' the data they are assigned to the parent scope.

Let's have a look at how this works:

```
def add_two(a, b): #Function Definition
    x = a + b

var = add_two(6, 9) #Function Call

print(var) # Checking if function is returning/outputting any data
print(type(var)) # Checking function's data type
```

Figure 5.7: Void Function

```
None
<class 'NoneType'>
```

Figure 5.8: Output

Here, the first thing to notice is that we can assign a function to a variable. We see that when we try to print the variable, it outputs 'None'. Furthermore, when we check its type, it outputs '<class 'NoneType'>'. This indicates that the function is not 'returning' any data.

On the other hand if we do this:
Here, 'return' is a Python keyword that breaks the function at that point and returns data that is assigned to it in the manner shown. This time when we print variable 'var', it prints '15' - which is exactly the value of x - instead of 'None'. Furthermore, when we check its type, it 'returns' '<class 'int'>'.

Note: I used 'return' instead of 'output' because type() is also a function, albeit a built-in one, that returns the data class type of data passed to it. From now on we will use 'return' more often and synonymously with 'output'.

```
def add_two(a, b): # Function Definition
    x = a + b
    return x

var = add_two(6, 9) # Assigning function

print(var) # Calling function through variable
print(type(var)) # Checking function's data type
```

Figure 5.9: Fruitful Function

```
15
<class 'int'>
```

Figure 5.10: Output

Additional data maybe added to 'return' statements by introducing comma's in-between i.e.

```
> def multiply_by_ten(a, b, c):
>       return a*10, b*10, c*10
```

However, y will also require additional variables for the data to be assigned to like this:

```
> var1, var2, var3 = multiply_by_ten(a, b, c)
```

Note: For outputting data to the console, it is necessary to use the print() function. 'return' statements do not print anything to the console, nor do function calls, variable calls or any sort of assignment operations. As a matter of fact, nothing does, except errors.

5.6 Built-in Functions

Like the functions we just talked about, Python has 68 built-in functions. They are called in a similar way and applied on different data types by passing the data as an argument. A few important ones are listed below, try them out, you may have used some before:

1. abs(*insert data*) - returns the absolute value of a decimal
2. bool() - converts data to boolean returning True or False
3. float() - converts an integer to float
4. help() - invokes the built-in python helping system over passed data.
5. input() - asks user/console for input
6. int() - converts float to integers
7. len() - returns length of a string
8. print() - prints data to user/console
9. round() - rounds float to nearest integer
10. type() - returns data's type

In an earlier section we also imported the math library to aid us with calculating the square root of a number. We did something like

```
>>> import math
>>> math.sqrt(16)
>>> 4
```

In this, sqrt() is one of the function, that belongs the math library. Similarly, we have other libraries i.e time, random etc that have several functions. The standard way to call a function from a library is:

```
>>> library_name.function_name()
```

5.7 Lab 4 - Exercises

Q1. Write a function named 'traffic_rules', that takes a parameter named 'signal_color', and prints the corresponding rule accordingly:

RED - "Stop"
YELLOW - "Get Set"
GREEN - "Go"

If input is invalid, it should print "Invalid Input". Make sure your program takes appropriate user input from the console as string and pass it to the function.

- - - - - SAMPLE- - - - -
Input: GREEN

```
Signal color: GREEN
Go
```

Input: BLUE

```
Signal color: BLUE
Invalid Input
```

Q2.(a) Write a function 'is_triangle' which takes parameters 'a', 'b' and 'c' as the lengths of a triangles side. The function is supposed to check and print whether the given side lengths can form a triangle or not.

For a triangle to be possible: According to the triangle inequality theorem, to form a triangle, the length of any two sides must be greater than the third. Hence, the following conditions must be true:

a + b > c
a + c > b
b + c > a

Make sure your program takes appropriate user input from the console as integers and passes it to the function.

- - - - - SAMPLE- - - - -
Input: 1 5 2

```
a: 1
b: 5
c: 2
These side lengths cannot form a triangle.
```

Input: 2 3 4

```
a: 2
b: 3
c: 4
These side lengths can form a triangle.
```

Q2.(b) Write a function 'is_right_triangle' which takes parameters 'a', 'b' and 'c' as the lengths of a triangles side. The function is supposed to check and print whether the given side lengths form a right-angled triangle or not. Hint: Builds up on part (a) and involves nested conditionals

According to Pythagoreans theorem, the sum of the square of the two shorter sides must equal the square of the longest side i.e. some combination of a,b and c should satisfy: $a^2 + b^2 = c^2$

Make sure your program takes appropriate user input from the console as integers and passes it to the function.

- - - - - SAMPLE- - - - -
Input: 2 3 4

```
a: 2
b: 3
c: 4
These side lengths cannot form a right-angled triangle.
```

Input: 3 4 5

```
a: 3
b: 4
c: 5
These side lengths can form a right-angled triangle.
```

Q3. Python has a built-in operation '*' to calculates the product of two numbers. Write a function named 'product' that takes two parameters 'a' and 'b', and returns the value of their product. You are not supposed to use the built-in operator, think loops!

- - - - - SAMPLE- - - - -
Input: 25, 6

```
a:25
b:6
150
```

Input: 7, 0

```
a:7
b:0
0
```

Q4. Python has a built-in operator '**' to calculates the power of a number. Write a function named 'power' that takes two parameters base and exponent, and returns the value of their power. You are not supposed to use the built-in operator, think loops!

Make sure your code makes appropriate considerations for negative bases and exponents. Your program should take appropriate user input from the console as integers and passes it to the function.

- - - - - SAMPLE- - - - -
Input: 4, 5
```
base: 4
exponent: 5
1024
```
Input: 0, 10
```
base: 0
exponent: 10
0
```
Input: 120, 0
```
base: 120
exponent: 0
1
```
Input: -2, 5
```
base: -2
exponent: -5
-0.03125
```

Q5. Your university uses the following grading scheme for the Computer Science program:

Marks	Grade	GPA
(94, 100]	A+	4.00
[90, 94)	A	4.00
[85, 90)	A-	3.67
[80, 85)	B+	3.33
[75, 80)	B	3.00
[70, 75)	B-	2.67
[67, 70)	C+	2.33
[63, 67)	C	2.00
[60, 63)	C-	1.67
[0, 60)	F	0.00

Some of your fellow colleagues are having trouble with how to go about calculating their CGPA (Cumulative Grade Point Average). To help your friends write a program for this purpose.

The CGPA is calculated by taking the cumulative sum of the gpa_obtained*course_credit_hours for every course taken, then dividing it by total achievable cumulative sum i.e. (4.00*course_credit_hours), and multiplying the result by 4

(a) Write a function named 'grade' that takes as parameter 'marks' and returns the grade as a string, as per the grading scheme given above.

- - - - - SAMPLE- - - - -
```
>>> grade(87)
'A-'
```

(b) Write a function named 'GPA' that takes as parameter 'grade' and returns the GPA (Grade Point Average) as an integer, as per the grading scheme given above.

- - - - - SAMPLE- - - - -

```
>>> GPA('A-')
3.67
```

(c) Write a function named 'CGPA' that takes as parameter 'num_of_courses' and returns the CGPA as an integer.

Make sure you program takes input the number of courses taken from the user and passes it to 'CGPA' which in-turn calls upon the helper functions made in part (a) and (b) to achieve the task at hand.

Hint: Within the function take inputs 'marks_obtained' and 'course_credits' from the user. Make use of loops for taking repetitive input for each course and keep
adding it to variables outside the loop for making the calculation. Some composition might help too!

- - - - - SAMPLE- - - - -

```
Total number of courses taken: 3
-----------------
Course 1
marks: 94
credit_hours: 3
-----------------
Course 2
marks: 86
credit_hours: 3
-----------------
Course 3
marks: 91
credit_hours: 4
---------------
CGPA: 3.901
```

VI

Functions- II

6. Functions - II

6.1 Default Arguments

We've seen how to pass arguments/parameters during function calls: either directly as values or through a variable.

We've also seen that the number of arguments passed during a function call need to be the same as the number of arguments defined during the function definition. Furthermore, the type of data passed needs to be consistent with how the arguments are used in the function body. Failing to do any of the former would result in errors.

Another thing we can do with arguments is assign them default values. This is done in the function definition. Once an argument is assigned a default value, it does not necessarily need to be passed a value during the function call.

However, if a value is still passed for the argument, then it over rides the default value. For example:

```
def calculate_two(a, b, operation = 'add'):
    if operation == 'add':
        return a + b
    elif operation == 'sub':
        return a - b

    else:
        return 'invalid operation'

print(calculate_two(4, 20))
print(calculate_two(424, 4, 'sub'))
```

Figure 6.1: Default Arguments

```
24
420
```

Figure 6.2: Output

There can be as many default arguments in a function definition as required, but it is necessary that all other non-default arguments be defined before any default arguments.

6.2 Composition

You may have heard about the idea of composition in Mathematics, with regard to functions, which is combining two functions by substituting the formula of one in the other.

For example, if $f(x) = 2x$ and $g(x) = 3x$, then the composition $f(g(x)) = 2(3x) = 6x$.

In context of programming, composition is very similar. It involves the passing of function calls of the same or different functions, as parameters.

```
def add_two(a, b):
    x = a + b
    return x

x = add_two(add_two(5, 5), add_two(4,20))

print(add_two(x, add_two(10,0)))
```

Figure 6.3: Composition

The final output is 44. So here's what's happening, first the add_two(5, 5) and add_two(4, 20) evaluate to 10 and 24 respectively. These two values are passed to the outer add_two function which evaluates to 34.

Next, this 34 is assigned to variable x and passed to the final add_two function call along with the inner function call add_two(10,0) which evaluates to 10. Together these two evaluate to 44.

6.3 Recursion

We've seen that we can pass function calls as parameters to the same or different function. Another interesting thing that we can do with functions is call the same or a different function within the code body of a function as long as they are defined before it in the code.

Lets see what happens if we call a function within itself:

```
def repeat_expression(exp):
    print(exp)
    print('Will this stop?')
    repeat_expression(exp)  # A recursive call.
    print('I don\'t think so')

repeat_expression('Habib University')
```

Figure 6.4: Infinite Recursion

```
Habib University
Will this stop?
Habib University
Will this stop?
Habib University
Will this stop?
Habib University
```

Figure 6.5: Output

So what we come across is an infinite sequence of two repeating statements, an iteration, quite similar to like something under a 'while' loop without a terminating condition. So what's happening here?

For the first run, it is us who call the function. This leads to execution of the first two print statements in the function body. The third statement in the function body is a call to the function itself. The function body re-executes itself at that point eventually reaching another function call. In a way the function keeps looping never reaching the last print statement.

If we were to visualize this execution line by line it would branch like this:

```
repeat_expression('Habib University') # Parent scope
   > print(exp)
   > print('Will this stop?')
   > repeat_expression(exp) # Child scope
        > print(exp)
        > print('Will this stop?')
        > repeat_expression(exp)
             > print(exp)
             > print('Will this stop?')
             > repeat_expression(exp)... # So on
```

Note: The indenting here shows a change in scope from parent to child

This behavior opens up the gate way for a very important and powerful computational tool called Recursion.

What we saw above was an example of infinite recursion. This is not exactly a desired state as we want our program to stop. For this we must define in our function a breaking condition, often called a 'base case'. This is how it works:

```
def lift_off(n):
    if n == 0: # Base case
        print('Lift Off!')
    else:
        print(n)
        return lift_off(n-1) # Recursive step

lift_off(3)
```

Figure 6.6: Finite Recursion

```
3
2
1
Lift Off!
```

Figure 6.7: Output

What's happening here is that until 'n' is not equal to 0 (base case) the function keeps printing 'n' and re-calls itself while decrementing the value of 'n' by 1 at each call. Eventually, assuming 'n' is a positive integer, its value will reach 0 and put an end to the recursive calls.

In essence, recursion is a 'divide and conquer' problem solving technique that involves breaking a problem and solving smaller instances of it to reach a solution. How is that so?

The base case is the point where the problem can be solved directly, its simplest case. The recursive step on the other hand helps reduce the problem to a smaller version of the original problem until it reaches the base case.

For example, consider the problem of multiplying two integers using repeated addition. Assuming both the integers are positive, the problem can be solved recursively as such:

```
def recur_multi(a, b):
    if b == 1:          # The Base Case
        return a
    else:
        return a+recur_multi(a, b-1) # The Recursive Step

print(recur_multi(5,5))
```

Figure 6.8: Recursive Solution

We know that if b equals 1, then a itself is the answer. This makes our base case. Meanwhile, our recursive case can be broken down and explained intuitively as follows:

$a * b = a + a + a + a + a...$ # b times
$= a + (a + a + a + a...)$ # (b-1) times
$= a + a * (b - 1)$ # recursive reduction

6.4 Towers of Hanoi

The problems we saw in section 6.3 could easily be solved using iteration too. So let's look at a more complex problem that is not easy solve iteratively, but becomes simple to solve when thought of recursively. This classic problem is known as 'Towers of Hanoi'.

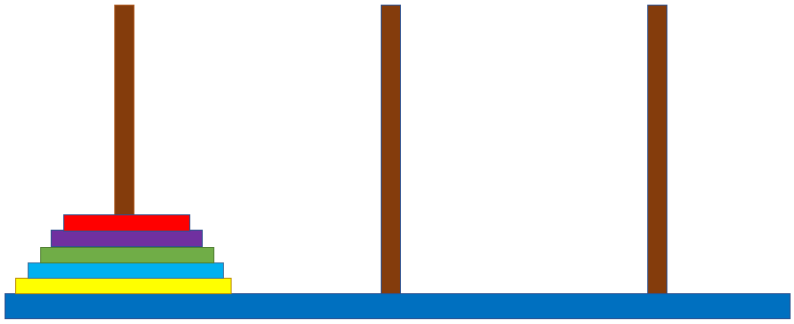


Figure 6.9: Towers of Hanoi

The legend is that there are 3 towers in a temple in Hanoi, Vietnam. One of the tower has 64 discs stacked over it such that no larger disc is above a smaller disc. The priests of this temple are to move this stack of discs from one tower to another, one discs at a time, such that no larger disc ends up at the top of a smaller during the process. According to legend, when the last disc is moved, the world will end.

The number of moves required to move a stack of size 'n' is given by $2^n - 1$. If it takes one second to make one move, then it will take $2^{64} - 1$ seconds or approximately 585 billion years to finish. So it is safe to conclude the legend is true.

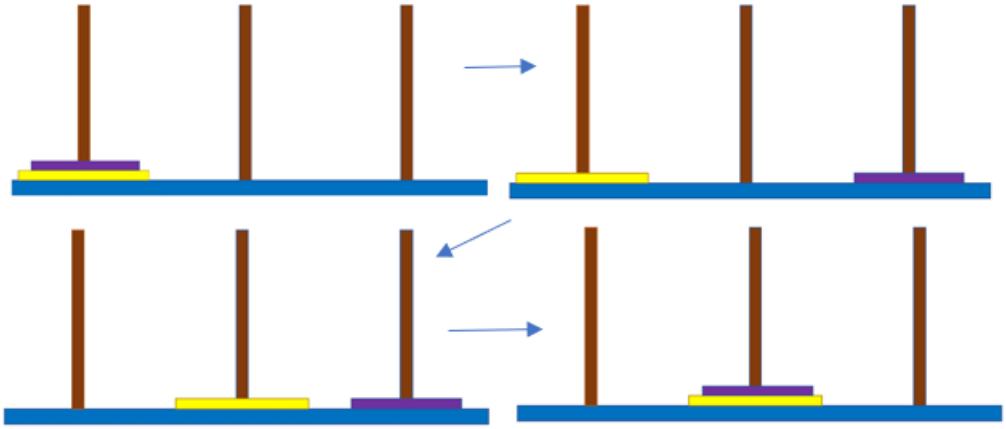


Figure 6.10: Working out a stack of size 2

Suppose A, B, and C are the three towers. A stack of size 2 is placed over tower A. Then the moves to shift the stack from A to B are:

1. Move a disc from A to C
2. Move a disc from A to B
3. Move a disc from C to B

The stack of size 2 was fairly simple to solve, however, when we get to a stack of size 3 it becomes a bit difficult to keep track of things.

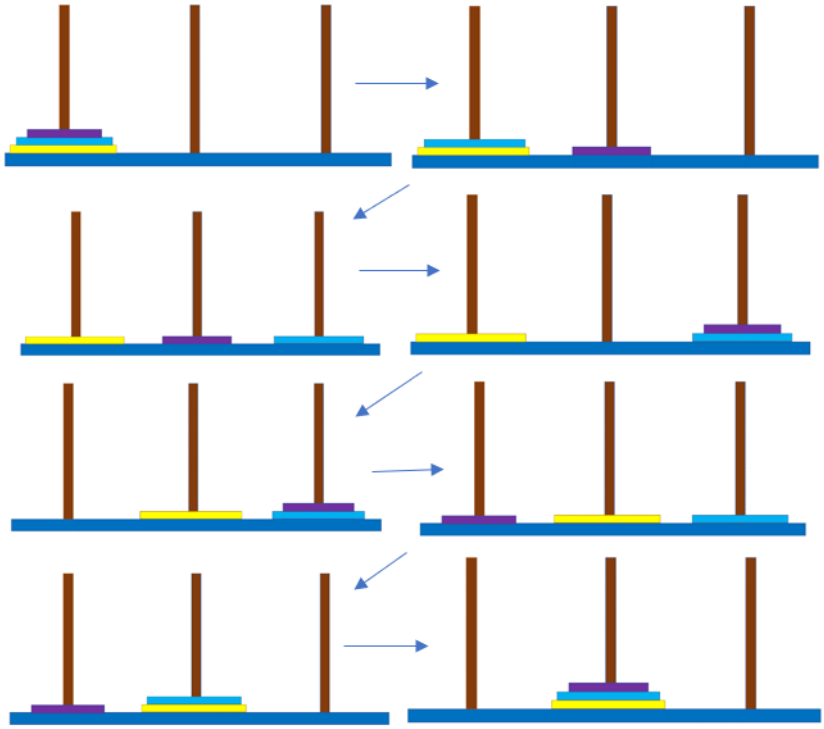


Figure 6.11: Working out a stack of size 3

Thinking of an iterative solution is not easy, so lets think recursively! What would be the base case? A stack of size 1, which would simple move from A to B.

What would be the recursive step? How could be break this to a smaller problem? The answer is simple, divide the stack to a stack of size of 2 by treating moving a stack of size n-1 to a spare tower, then moving the last left-over disc to the target tower, and then again moving the stack of size n-1 from the spare to the target tower.

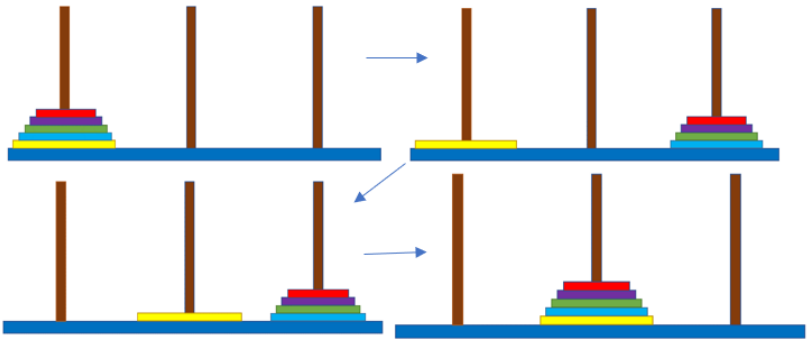


Figure 6.12: Working out a stack of size 3

The code for the recursive solution is as follows:

```
def Hanoi(n, src, tar, spare):
    if n == 1: # Base case
        print('Move from', src,'to', tar) # print Move
    else:
        Hanoi(n-1, src, spare, tar) # Move stack of size n-1 from source to spare
        Hanoi(1, src, tar, spare) # Move left over single disc from source to target
        Hanoi(n-1, spare, tar, src) # Move stack of size n-1 from spare to taget

Hanoi(3, 'A', 'B', 'C')
```

Figure 6.13: Recursive Solution to Tower of Hanoi

```
Move from A to B
Move from A to C
Move from B to C
Move from A to B
Move from C to A
Move from C to B
Move from A to B
```

Figure 6.14: Output

6.5 Lab 5 - Exercises

Q1.(a) Write a function 'is_prime_iter' that takes as parameter and returns a bool value for whether the integer is a prime number or not (True or False). Use iteration to solve this problem.

- - - - - SAMPLE- - - - -
Input: 113, 121, 131

```
>>> is_prime_iter(113)
True
>>> is_prime_iter(121)
False
>>> is_prime_iter(131)
True
```

Q1.(b) Write a function 'is_prime_recur' that takes as parameter and returns a bool value for whether the integer is a prime number or not (True or False). Use recursion to solve this problem. Hint: Make use of default arguments!

- - - - - SAMPLE- - - - -
Input: 113, 121, 131

```
>>> is_prime_recur(113)
True
>>> is_prime_recur(121)
False
>>> is_prime_recur(131)
True
```

Q2. Factorial of any number n is represented by n! where
$n! = 1*2*3*....*(n-1)*n$. i.e.
$5! = 1*2*3*4*5 = 120$
$4! = 4*3*2*1 = 24$
$3! = 3*2*1 = 6$

Write a function named 'fact' that takes the parameter 'n' as an integer and returns the factorial of 'n'. Use recursion to solve this problem.

Note: $1! = 1$ and $0! = 1$

- - - - - SAMPLE- - - - -
Input: 5

```
>>> fact(3)
6
>>> fact(4)
24
>>> fact(5)
120
>>> fact(6)
720
```

Q3. Python has a built-in operator '**' to calculates the power of a number. Write a function named 'power' that takes two parameters base and exponent, and returns the value of their power. You are not supposed to use the built-in operator, think recursively!

Make sure your code makes appropriate considerations for negative bases and exponents.

- - - - - SAMPLE- - - - -

```
>>> power_recur(2,5)
32
>>> power_recur(-4,5)
-1024
>>> power_recur(-4,6)
4096
>>> power_recur(-2,-5)
-0.03125
>>> power_recur(0, 2)
0
>>> power_recur(12, 0)
1
```

Q4.(a) Write a function named 'count_up_even' that takes parameters 'a' and 'b' and prints all even numbers in-between (a and b inclusive) in ascending order. Use recursion to solve this problem.

- - - - - SAMPLE- - - - -

```
>>> count_up_even(0,0)
0
>>> count_up_even(2,3)
2
>>> count_up_even(1,5)
2
4
>>> count_up_even(-2, 6)
-2
0
2
4
6
```

Q4.(b) Write a function named 'count_down_odd' that takes parameters 'a' and 'b' and prints all odd numbers in-between (a and b inclusive) in descending order. Use recursion to solve this problem.

- - - - - SAMPLE- - - - -

```
>>> count_down_odd(1,1)
1
>>> count_down_odd(4,1)
3
1
>>> count_down_odd(6,-3)
5
3
1
-1
-3
```

Q5. The Fibonacci sequence is a series of numbers where a number is found by adding up the two numbers before it. Starting with 0 and 1, the sequence goes 0, 1, 1, 2, 3, 5, 8, 13, 21, 34....

Write a function 'fib' that takes input as parameter a positive integer 'n' and prints the nth term of the sequence. Use recursion to solve this problem.

- - - - - SAMPLE- - - - -

```
>>> fib(1)
0
>>> fib(2)
1
>>> fib(3)
1
>>> fib(5)
3
>>> fib(8)
13
>>> fib(13)
144
```

VII

Strings - I

Python Strings

7. Strings - I

7.1 What are Strings?

A string is a Python data type. It is a sequence of characters. Strings are most often used when data or instructions need to be output to the console or external data needs to be read.

```
def Strings():
    print('This is a string')
    x = "This is also a string"
    v = """So is this: 1234555
12313435453643#&$&"""
    print('')
    print(v)
    print('')
    return 'Returning a string'

print(Strings())
```

Figure 7.1: Strings

```
This is a string

So is this: 1234555
12313435453643#&$&

Returning a string
```

Figure 7.2: Output

7.2 Indexing

An interesting feature of strings is that with each position in the sequence there is an associated integer or index starting from 0. The 0th index is associated with the 0th position. Hence, if there are 8 positions in a string, then the index ranges from 0 to 7.

Indexes can also range in negative integers, starting from -1, which represents the last position. Hence, if there are 8 positions in a string, then the reverse index ranges from -1 to -8.

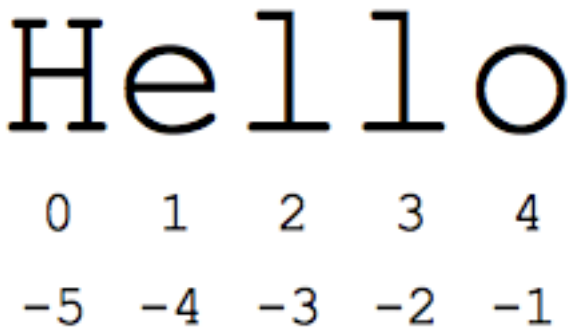


Figure 7.3: Indexing 101

Indexing is vital to string manipulation (extracting information from or storing information in strings). In particular, indexes help extract a character from the corresponding index position.

The general form for extracting a character from

```
>>> quote_1 = 'Argue for your limitations, and sure enough, they are yours'
>>> letter = quote_1[0]
>>> print(letter)
A
>>> print(quote_1[5]) # Empty space

>>> print(qoute_1[-1]) # Reverse indexing
s
>>> print(qoute_1[50]) # index out of range
Traceback (most recent call last):
File "<pyshell#2>", line 1, in <module>
x[13]
IndexError: string index out of range
```

7.3 Length of a String

It often required that we have the length of a string as an integer for manipulation. Counting manually and hard-coding is not feasible, nor an option most of the time.

For this purpose we have a built-in function 'len()' which takes as parameter a string and returns its length as an integer.

```
>>> string = 'Think'
>>> len(string)
```

```
5
```

It is worth noting that the maximum index range is always one less than the length of the string. For example, in the example above the index ranges from 0 to 4.

7.4 Slicing

Another important feature indexing opens up is the ability to slice strings. This allows us to extract characters from more than a single position in the string sequence.

The general form for doing this is:

```
>>> some_string[x:y:z]
```

- 'x' is index it starts slicing from and is inclusive. By default this is the first index position.
- 'y' is the index at which it ends the slicing and 'y' itself is not inclusive. By default this is the last index + 1 (Since the last index is not inclusive otherwise).
- 'z' is the step it takes between indexes while slicing from x to y. By default this is 1.

```
>>> song = 'Lost in the Echo - Linkin Park'
>>> artist = song[19:31]
>>> print(artist)
Linkin Park
>>> artist = song[19:len(song)] # Note: Last index isn't inclusive, so we
    don't subtract one from the length
>>> print(artist)
Linkin Park
>>> artist = song[19:]
>>> print(artist)
Linkin Park
```

Note: The first three ways of slicing yield the same result.

```
>>> quote_2 = 'What the caterpillar calls the end of the world, the master
    calls a butterfly'
>>> print(quote_2[::2]) # skips 1 position.
SaeOdt ai oi
```

We can also reverse slice a string by using a negative step value.

```
>>> reverse_quote = quote_2[::-1]
>>> print(reverse_quote)
ylfrettub a sllac retsam eht ,dlrow eht fo dne eht sllac rallipretac eht tahW
```

7.5 Immutability

Strings are immutable. This means they cannot be changed or updated in anyway. A variable maybe updated with an altogether new string, however, the original can never be changed or updated.
For example, suppose the following string:

```
>>> str_1 = 'The simplest things are often the truest.'
```

There is no method using which we can change 'o' in this string to so some other character. What we can do is update the variable 'str_1' with a new string.

```
>>> str_1 = 'The simplest things are aften the truest.' # This is a new
    string.
```

The idea of immutability will make more sense once we come across mutable data structures like lists.

7.6 Concatenation

While we may not be able to change or update strings, however, we can concatenate them. This means we can stitch them together to form a new string.

```
>>> firstname = 'Habib'
>>> lastname = 'University'
>>> full_name = firstname + ' ' + lastname # Concatenating
>>> print(full_name)
Habib University
>>> id = 2021
>>> info = fullname
>>> info += ' ' + str(id) # Concatenating and converting integer to string.
>>> print(info)
Habib University 2021
```

7.7 Traversal

In the chapter on loops we learned about how 'for' loops can be used to iterate over data using indexes or directly over the data itself. This is called traversal.

```
string = 'Happiness is the reward we get for living to the highest right we know'

# Direct Traversal
for char in string:
    print(char, end = '') # end = '' makes each character print on the same line.

print('')
# Traveral using indexes
for i in range(0, len(string)):
    print(string[i], end = '')
```

Figure 7.4: Traversal

Traversal over strings is essential for being able to read and manipulate strings. For example, unlike Python, many languages don't have a built-in function for counting and returning the length of a string. Using traversal we could write a function for counting the length of a string.

```
def length(some_str):
    count = 0
    for char in some_str:
        count += 1

    return count
```

Figure 7.5: Length Function

```
>>> length('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
26
```

Figure 7.6: Output

Similarly, it could also be used for counting the number of times a specific character appears in a string.

```
def count_char(char, some_str):
    count = 0
    for i in range(0, len(some_str)): # Using indexing instead of
        if some_str[i] == char:       # direct traversal
            count += 1

    return count
```

Figure 7.7: Character Count Function

```
>>> count_char('a', '''There are no mistakes. The events we
bring upon ourselves, no matter how unpleasant, are necessary
in order to learn what we need to learn; whatever steps we take,
they're necessary to reach the places we've chosen to go''')
15
```

Figure 7.8: Output

The ability to iterate over strings vastly increases the scope of how they can be used and manipulated.

7.8 Lab 6 - Exercises

Q1. Write a function named 'replacer' that takes as parameters 'frm','to', and 'some_str' as strings and returns a new string with a character replaced by a new one.

- - - - - SAMPLE- - - - -

```
>>> print(replacer('i', 'o', 'Habib University'))
Habob Unoversoty
```

Q2. Write a function 'find_index' that takes as parameter 'char' and 'string' as strings, and returns as an integer the first index where the character is found in the string. If the character is not found in the string, return -1

You cannot use built-in string functions.

Note: The length of character cannot be less than or greater than 1, then return 'Error: bad argument'. Length of character must be 1.

- - - - - SAMPLE- - - - -

```
>>> find_index('a', 'safe')
1
>>> find_index('b', 'safe')
-1
>>> find_index('ab', 'banana')
"Error: bad argument. Length of 'char' must be 1"
```

Q3. Many programming languages do not have a built-in slice methods like Python. Write a function named 'slicer' that takes parameters 'string', 'start', 'end', and 'step'. It returns the sliced part of the string as per the parameters given.

Make sure the slicer function operates the same as the built-in slice method.'start' and 'end' will not have default values, however, give an appropriate default value for 'step'.

Also make considerations for if the 'step' value is negative.

- - - - - SAMPLE- - - - -

```
>>> data = 'PFun? or not so fun?'
>>> print(slicer(data, 0, 5))
PFun?
>>> print(slicer(data, 0, len(data), 2))
Pu?o o ofn
>>> print(slicer(data, 0, len(data), -1))
?nuf os ton ro ?nuFP
>>> print(slicer(0, len(data), -2))
Error: bad argument.
>>> print(slicer(data, 0, len(data), -2))
?u stnr nF
>>> print(slicer(data, -9, -1))
Error: bad argument.
```

Q4. Write a function named 'remove_redundant' that takes a parameter 'some_str' as a string and returns a new string in which no character of the original string appears more than once excluding empty spaces.

Hint: Use nested loops

- - - - - SAMPLE- - - - -

```
>>> print(remove_redundant('zabaz123243516257@$$%!@*%@!'))
zabl234567@$%!*
>>> print(remove_redundant('aaaabbbaaccbdddee ffeegggjjk kllmmnooaaee'))
abcde fgjk lmno
```

---

*Q5. Write a function named 'squares' that takes a parameter 'size' and returns as a single string, four squares of the shape shown below.

Hint: You might need to use concatenation, '\n' for starting a new string line and loops.

- - - - - SAMPLE- - - - -

```
>>> print(squares(5))
*****
*****
*****
*****
*****

*****
*   *
*   *
*   *
*****

*****
**  *
* * *
*  **
*****

*****
*  **
* * *
**  *
*****
```

```
>>> print(squares(4))
****
****
****
****

****
*  *
*  *
****

****
** *
* **
****

****
* **
** *
****
```

```
>>> print(squares(3))
***
***
***

***
* *
***

***
***
***

***
***
***
```

```
>>> print(squares(2))
**
**

**
**

**
**

**
**
```

**Q6. You are working for your universities IT department over the summers. The admissions office has given the IT department the name of students getting enrolled this year. The current number of students enrolled is 789.

To make things easier. You are assigned the task of writing an algorithm that takes input a string of comma separated names of students and returns a string of comma separated emails of the students as per their names in the following format:

Name: 'Ahsan Qadeer'

Student Number: 3568
Email: aq03568@st.habib.edu.pk

These will then be added to the universities email server as new emails.

- - - - - SAMPLE- - - -

```
>>> print(generate_email('''waqar shah,sarwan saleem,abdullah abbas,
hashim siddique,hamza khan,adil junaid,zainab habiby,marium hasnain'''))
ws00790@habib.edu.pk,ss00791@habib.edu.pk,aa00792@habib.edu.pk,
s00793@habib.edu.pk,hk00794@habib.edu.pk,aj00795@habib.edu.pk,zh00796@habib.edu.pk
```

VIII Strings - II

Python Strings

8. Strings - II

8.1 'in' Operator

Like 'for', 'while', 'print' etc, 'in' is also a Python keyword. You may have noticed it being used for a 'for' loop i.e.

```
>>> var = 'cat'
>>> for char in var:
      print(char)
c
a
t
```

The 'in' operator is usable over any data type that is iterable. It accesses each element of the data through a background loop. It is often used for checking the presence of a single or group elements in the data. For example,

```
>>> 'c' in 'abcdefg'
True
>>> 'var' in 'thevariable'
True
>>> 'vara' in 'thevariable'
False
>>> not 'the' in 'thevariable'
False
```

This makes it very helpful when dealing with strings.

8.2 Built-in String Functions

Python has several built-in String functions or methods. These make string reading and manipulation easier. A few important ones are listed below. The complete list can be found by executing the following command in the Python/IDLE shell:

```
>>> help(str)
```

Function	Description
some_str.upper()	Returns a string with all alphabets of the original string converted to uppercase
some_str.lower()	Returns a string with all alphabets of the original string converted to lowercase
some_str.index(x)	Returns as integer the first index where the character is found or the string starts or else returns -1
some_str.count(x)	Returns as integer the number of times a substring occurs in a string.
some_str.isalnum()	Returns True if string only has alphabets or numbers.
some_str.isalpha()	Returns True if string only has alphabets.
some_str.isnumeric()	Returns True if string only has numbers.
some_str.strip(x)	Returns a string with all leading and lagging character's 'x' are stripped from the original string. If no character is given as parameter then white spaces are stripped.

Sample outputs in Python/IDLE shell:

```
>>> 'abc'.upper()
'ABC'
>>> 'abcdef'.index('e')
4
>>> 'abcdefaghk'.index('agh')
6
>>> 'abcdefabcabc'.count('abc')
3
>>> '123fourfivesix'.isalnum()
True
>>> 'five'.isnumeric()
False
>>> '**1123**'.strip('*')
'1123'
>>> '  11 23  '.strip()
'11 23'
```

8.3 ASCII Values

In Python each character has an associated ASCII value. This value can be obtained for any character by using the built-in Python function 'ord()' and passing the character as a string to the function i.e.

```
>>> ord('A')
65
```

Similarly, if a character's ASCII value is known, then the character can be obtained as a string by passing its ASCII value as an integer to the built-in Python function 'chr()' i.e.

```
>>> chr(65)
'A'
```

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Figure 8.1: ASCII Table

8.4 Files

The data we have been dealing with so far either lies within the program or is taken as input from the user. However, We are also often required to read/write data from/to files.

Python has a built-in method to do this. The file is accessed using the keyword 'open' in the following manner:

```
>>> file_data = open('file_name', 'mode')
```

This opens or loads the data present in the file into a program as an iterable data type in which each element represents a line in the file as a string. The data can then be accessed by using a for loop on it.

There the three basic modes are:
1. 'r' - is the read only mode, in which data cannot be written to the file. This is also the default mode.
2. 'w' - is the write only mode, in which data can only be written to the file, but not read.
3. 'r+' - is read and write mode, both operations can be performed.

In case no file already exits and it is opened using the open() method in 'w' mode, then it will automatically serve the purpose of creating a new file.

Let's look at an example of how data can be read and written to a file:

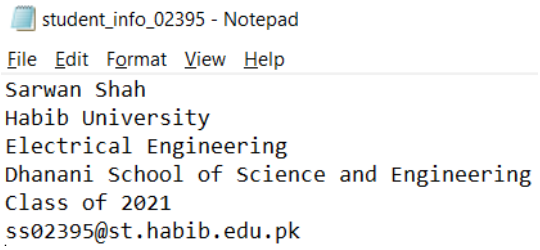


Figure 8.2: File Before Writing

```
file_data = open('student_info_02395.txt', 'r+') #Opening file

print('Current Info on Student:')
for line in file_data:
    print(line.strip()) # Removing extra spaces and printing each line.

file_data.write('Courses Taken: PFun, DLD, Calc 1, Rhetcom') # Writing data to file

file_data.close() # Closing file
```

Figure 8.3: Reading the file

```
Sarwan Shah
Habib University
Electrical Engineering
Dhanani School of Science and Engineering
Class of 2021
ss02395@st.habib.edu.pk
```

Figure 8.4: Output

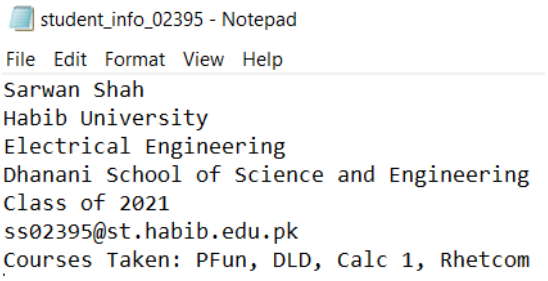


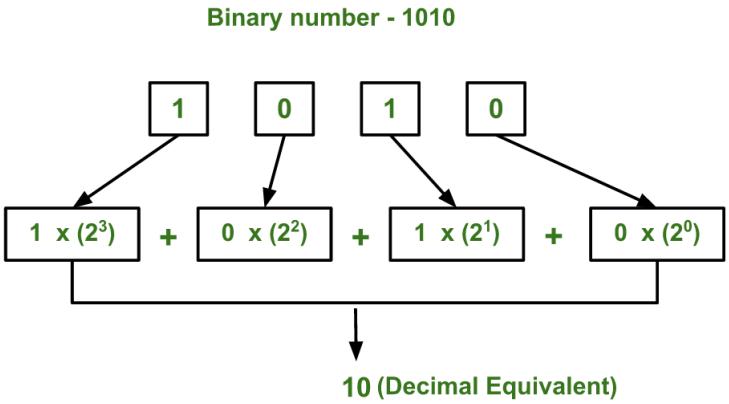
Figure 8.5: File After Writing

It is worth-noting that it is important to close the file using '.close()' function oNow that we know how to read and write data, we can use string manipulation techniques to extract and add required information.

8.5 Lab 7 - Exercises

Q1.(a) Write a function named 'binary_to_deci' that takes as parameter 'n', as a string, representing a binary number and returns as an integer its decimal value.

Binary to decimal conversion works in the following way:

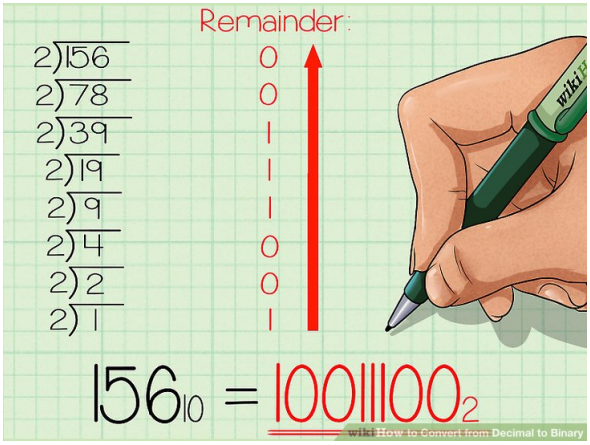


- - - - - SAMPLE- - - - -

```
>>> print(binary_to_deci('1101'))
13
>>> print(binary_to_deci('11'))
3
>>> print(binary_to_deci('100'))
4
```

Q1.(b) Write a function named 'deci_to_binary' that takes as parameter 'n', as a positive integer, representing a decimal number and returns as a string its binary representation.

Decimal to binary conversion works in the following way:



- - - - - SAMPLE- - - - -

```
>>> print(deci_to_binary(13))
1101
>>> print(deci_to_binary(1))
1
>>> print(deci_to_binary(5))
101
```

Q2.(a) Making use of ASCII values, write a function named 'is_num' that takes a parameter 's' as a string and returns True if and only if all character's in the string are numeric. Do not use any built-in Python string methods.

- - - - - SAMPLE- - - - -

```
>>> print(is_num('78nine'))
False
>>> print(is_num('342693'))
True
>>> print(is_num(''))
False
```

Q2.(b) Making use of ASCII values, write a function named 'is_alphanum' that takes a parameter 's' as a string and returns True if and only if all characters in the string are either alphabets or numbers. Do not use any built-in Python string methods.

- - - - - SAMPLE- - - - -

```
>>> print(is_alphanum('123456seveneightnine'))
True
>>> print(is_alphanum('1234'))
True
>>> print(is_alphanum('ABCD'))
True
>>> print(is_alphanum('ABCD.1234'))
False
>>> print(is_alphanum(''))
False
```

Q2.(c) Making use of ASCII values, write a function named 'is_lower' that takes a parameter 's' as a string and returns a new string with all the characters in lower case. Do not use any built Python string methods.

- - - - - SAMPLE- - - - -

```
>>> print(is_lower('Henlo FRENS'))
henlo frens
>>> print(is_lower('say_My_NAME'))
say_my_name
```

Q3.(a) A palindrome is a word or phrase that reads the same backward and forward. For example: 'No lemon, no melon' - 'nolem on, nomelon'

Write a function named 'is_palindrome' that takes a parameter 's' as string and returns True if and only if it is a palindrome. Ignore special characters.

- - - - - SAMPLE- - - - -

```
>>> print(is_palindrome('Eva, can I see bees in a cave?'))
True
>>> print(is_palindrome('Red rum, sir, is murder'))
True
>>> print(is_palindrome('Engineering is great'))
False
```

Q3.(b) An anagram is a word or phrase formed by rearranging the letters of a different word or phrase. For example: 'The Morse Code' and 'Here comes dot'.

Write a function named 'are_anagrams' that takes as parameters, two strings, 'a' and 'b', and returns True if and only if the words are anagrams of each other.

- - - - - SAMPLE- - - - -

```
>>> print(are_anagrams('O, Draconian devil!', 'Leonardo da Vinci'))
True
>>> print(are_anagrams('Oh, lame saint', 'The Mona Lisa'))
True
>>> print(are_anagrams('The eyes', 'They see'))
True
>>> print(are_anagrams('Little Dogs', 'Dig Lit Logs'))
False
```

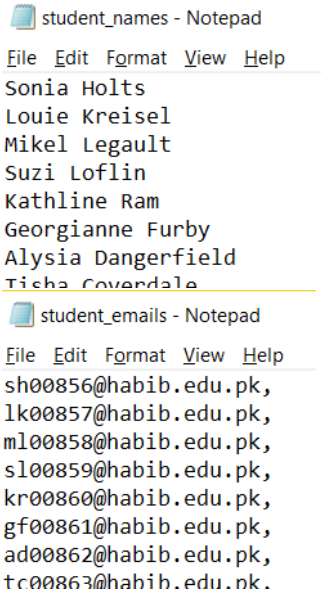
**Q4. You are working for your universities IT department over the summers. The admissions office has given the IT department the file containing the name of all the students getting enrolled this year. Each students name is on a separate line. The current number of students enrolled is 856.

To make things easier you are assigned the task of writing an algorithm that reads the file of student names and generates a new file named 'student_emails.txt' that contains the emails of all students on separate lines. Student emails are based on the following format:

Name: 'Ahsan Qadeer'
Student Number: 3568
Email: aq03568@st.habib.edu.pk

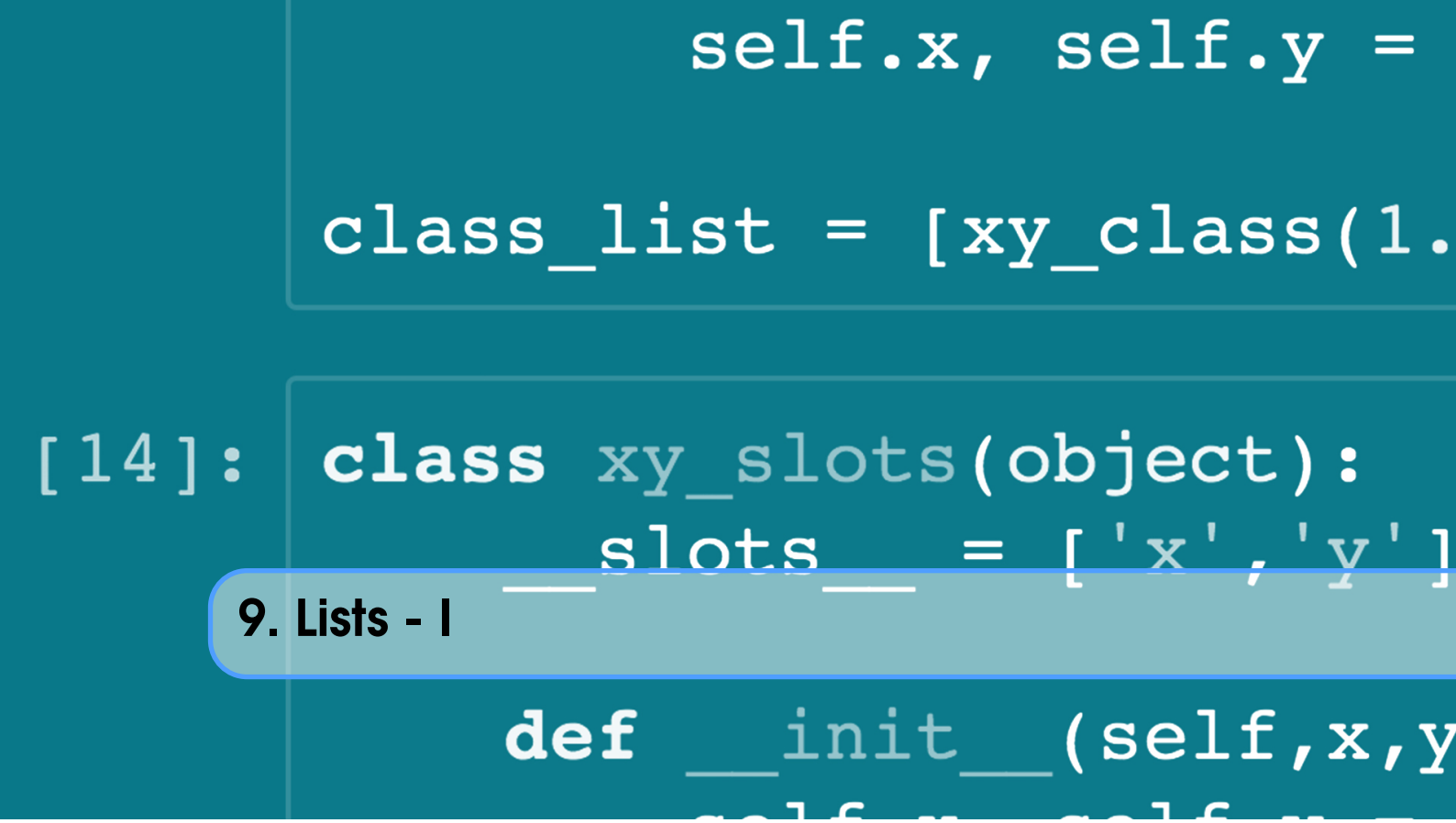
These will then be added to the universities email server as new emails. The student names file can be downloaded from here.

- - - - - SAMPLE- - - - -



IX

Lists - I



9. Lists - I

In this section we will look at a new and very essential Python data type called lists. Lists are described better as a data structure rather than a data type because they within themselves can carry elements of multiple data types including nested lists.

The ability of lists to store and handle multiple data types at once makes them an essential tool when it comes to programming.

9.1 What are lists?

A list is a Python data type. Similar to strings, lists are a sequence of elements instead of characters. These elements can be of the same or different data type, or even a nested list (list within a list). Each element is separated by a comma in-between, for example:

```
>>> variable_bool = True
>>> my_list = ['this a string in a list', 420, ['This a string in a list
    within list'], 'blaze it', variabl_bool] # Initializing a list
>>> print(my_list)
['this a string in a list', 420, ['This a string in a list within list'],
    'blaze it', True]
```

9.2 Length of a list

Like strings, the length of a string is the number of elements the list has instead of character. It is often required for list manipulation purposes. The built-in function 'len()' can be applied to lists by passing it as the parameter.

```
>>> seven_sins = ['Wrath', 'Gluttony', 'Greed', 'Envy', 'Sloth', 'Pride',
    'Lust']
>>> print(len(seven_sins))
7
```

9.3 List Indexing

List indexes work the same as string indexes. The 0th index is associated with the 0th position and the element it carries. Hence, if there are 8 elements in a string, then the indexes range from 0 to 7.

Similarly, indexes can also range in negative integers, starting from -1, which represents the last position. Hence, if there are 8 positions in a list, then the reverse indexes range from -1 to -8.

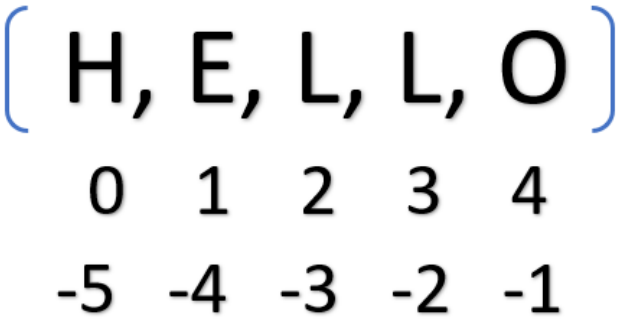


Figure 9.1: List Indexing 101

```
>>> my_list = ['Fire', 'Air', 'Water', 'Earth']
>>> print(my_list[1]) # Mind the brackets
'Air'
>>> print(my_list[len(list) - 1]) # Why did we subtract 1?
'Earth'
>>> print(my_list[-4])
'Fire'
```

9.4 List Slicing

List slicing also works the same as string slicing. It allows us to extract elements from more than one position in the list sequence by dividing the original list into a smaller list sequence.

The general form for doing this is:

```
>>> some_list[x:y:z]
```

- 'x' is index it starts slicing from and is inclusive. By default this is the first index position.
- 'y' is the index at which it ends the slicing and 'y' itself is not inclusive. By default this is the last index + 1 (Since the last index is not inclusive otherwise).
- 'z' is the step it takes between indexes while slicing from x to y. By default this is 1.

For example:

```
>>> student_info = ['Hamza', 'Khan', 3596, 'Electrical Engineering', 2021]
>>> name = student_info[0:2]
>>> print(name)
['Hamza', 'Khan']
```

```
>>> class = student_info[3:len(list)] # Last index isn't inclusive, so don't
    subtract 1
>>> print(class)
['Electrical Engineering', 2021]
>>> print(student_info[3:])
['Electrical Engineering', 2021]
>>> print(student_info[::2])
['Hamza', 3596, 2021]
```

We can also reverse slice a list by using a negative step value.

```
>>> reverse_info = student_info[::-1]
>>> print(reverse_info)
[2021, 'Electrical Engineering', 3595, 'Khan', 'Hamza']
```

If a list has a nested-lists, then its elements can also be accessed using slicing

```
>>> lst = [['a','b','c']['d','e','f'],['g','h','i']]
>>> print(lst[1][2])
'f'
```

9.5 Mutability of lists

Lists are a mutable data type. This means their elements can be changed and updated. While the variable associated with the list maybe updated with an altogether new list, the original list can also be changed or updated.

For updating an existing element we use indexing along with the assignment operator in the following way:

```
>>> grade = ['PFun', 'Undecided']
>>> grade[2] = 'A+'
>>> print(grade)
['PFun', 'A+']
```

A new element can be added to the existing list in the following way:

```
>>> grade.append('94.56 %')
>>> print(grade)
['PFun', 'A+', '94.56 %']
```

Note: New elements will always be added towards the end/top of the list and multiple elements cannot be appended nor updated to the list in a single call.

9.6 Concatenation of lists

Like strings, lists too can be concatenated. This means we can stitch/add them together to form new lists.

```
>>> first_name = ['Ali']
>>> last_name = ['Hasan']
>>> name = first_name + last_name
>>> print(name)
['Ali', 'Hasan']
```

9.7 List traversal

List are an iterable data type. This means we can use loops to iterate over the data in them, either using indexes or directly. The syntax is the same as for strings.

```
lst = ['one', 'two', 'three', 4, 5, 6]

# Direct Traversal
for elem in lst:
    print(elem, end = '')

# Traversal using indexes
for i in range(0, len(lst)): # Last index isn't inclusive
    print(lst[i]) # How is this different from the print statement above?
```

Figure 9.2: List traversal

A list within a list can also be traversed using nested loops:

```
lists = [[1,2],[3,4],[5,6]]

for lst in lists:
    for element in lst:
        print(element)
```

Figure 9.3: Traversing nested lists

```
123456
```

Figure 9.4: Output

9.8 Lab 8 - Exercises

Q1.(a) It often required that programs need to be checked and guarded against invalid inputs.

Write a function 'check_types' that takes as parameter a list 'lst' and returns a list of all the data types that were present in the list that was passed as a parameter. Your function should also include guards against invalid arguments.

- - - - - SAMPLE- - - - -

```
>>> print(check_types([]))
[]
>>> print(check_types([2,3,5]))
['int']
>>> print(check_types([5,6,'six',['eight','nine'],True]))
['int', 'str', 'list', 'bool']
>>> print(check_types(['hello',[2,[False, [3.5]]],'world']))
['str', 'list', 'int', 'bool', 'float']
>>> print(check_types('this is not right'))
Error: Bad argument. Function 'check_types' only accepts lists.
>>>
```

Q2.(a) Python has a built-in function 'max()' that can be applied to a list and returns the highest number in the list, provided all the elements are numeric. The numbers in the list can all either be integers and floats, or strings.

For example:

```
>>> max([2,10,8,11.5,6])
11.5
>>> max(['2','9.25','9.5','8'])
'9.5'
```

Write your own function 'user_max()' that can do the same. Your function should include guards against invalid arguments.

- - - - - SAMPLE- - - - -

```
>>> print(user_max([2,4,10,8]))
10
>>> print(user_max(['2.5','3.5','3.25']))
3.5
```

Q2.(b) Similarly, there is also a built-in function 'min()' that does the opposite. Write your own function 'user_min()' that takes as parameter a list: 'lst', and returns the lowest number in the list, provided all elements are numeric.

- - - - - SAMPLE- - - - -

```
>>> print(user_min([5,8,3,4]))
3
>>> print(user_min([-5,6,2,-8,0]))
-8
```

Q3.(a) Write a function named 'breakdown' that takes a parameter 'n' as an integer and returns a list containing all the units, tens, thousands etc that make up the number.

- - - - - SAMPLE- - - - -

```
>>> breakdown(100)
[100]
>>> breakdown(1256)
[1000, 100, 100, 10, 10, 10, 10, 10, 1, 1, 1, 1, 1, 1]
>>> breakdown(13202)
[10000, 1000, 1000, 1000, 100, 100, 1, 1]
>>> breakdown(-2342)
[-1000, -1000, -100, -100, -100, -10, -10, -10, -10, -1, -1]
```

Q3.(b) You are working as a programmer at a bank. You are required to write a function 'atm_out' for the bank's ATM machines that takes as parameter a positive integer 'amount' and returns a list containing list-pairs of the note and its quantity that needs to be outputted by the ATM to meet the input amount needs.

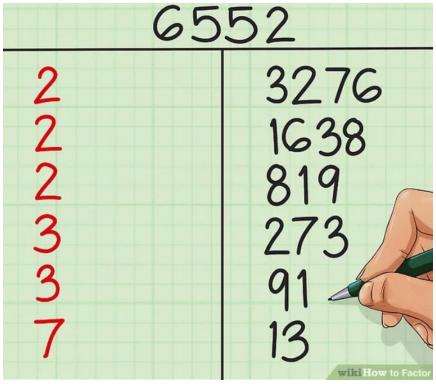
The minimum input amount must be greater than or equal to 500.

- - - - - SAMPLE- - - - -

```
>>> atm(5250)
[[1000, 5], [100, 2], [50, 1]]
>>> atm(1570)
[[1000, 1], [500, 1], [50, 1], [10, 2]]
>>> atm(480)
'Error: the minimum input amount is 500'
```

Q4.(a) A special number is a number that is equal to the sum of its prime factors.

"A prime number can only be divided by 1 or itself, so it cannot be factored any further! Every other whole number can be broken down into prime number factors. It is like the Prime Numbers are the basic building blocks of all numbers." - Math is fun really?



Write a function 'prime_factors' that takes as parameter 'num', a positive integer, and returns a list containing its prime factors.

- - - - - SAMPLE- - - - -

```
>>> prime_factors(6552)
[2, 2, 2, 3, 3, 7, 13]
>>> prime_factors(256)
[2, 2, 2, 2, 2, 2, 2, 2]
>>> prime_factors(1567)
[1567]
```

Q4.(b) Write another function named 'special_numbers' that takes as parameter a positive integer 'num' and returns a list of all the specials numbers from 0 to that number inclusive.

Use your 'prime_factors' function from the previous part in this function.

- - - - - SAMPLE- - - - -

```
>>> special_num(25)

[0, 2, 3, 4, 5, 7, 11, 13, 17, 19, 23]
>>> special_num(50)

[0, 2, 3, 4, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43
, 47]
>>> special_num(150)

[0, 2, 3, 4, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43
, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 1
07, 109, 113, 127, 131, 137, 139, 149]
```

Q5.(a) Write a function 'count_char' that takes as parameter 'some_str' as a string and returns a list containing nested-list pairs of the count of each character, including special characters and spaces.

- - - - - SAMPLE- - - - -

```
>>> count_char('This is easier than it looks')
[['t', 1], ['h', 2], ['i', 4], ['s', 4], [' ', 5], ['e', 2], ['a', 2], ['r',
1], ['t', 2], ['n', 1], ['l', 1], ['o', 2], ['k', 1]]
>>> count_char('Approach this problem step by step!')
[['a', 1], ['p', 5], ['r', 2], ['o', 2], ['c', 1], ['h', 2], [' ', 5], ['t',
3], ['i', 1], ['s', 3], ['b', 2], ['l', 1], ['e', 3], ['m', 1], ['y', 1], ['!
', 1]]
>>> count_char('Any built in functions you could use?')
[['a', 1], ['n', 4], ['y', 2], [' ', 6], ['b', 1], ['u', 5], ['i', 3], ['l',
2], ['t', 2], ['f', 1], ['c', 2], ['o', 3], ['s', 2], ['d', 1], ['e', 1], ['?
', 1]]
>>> count_char(['Don\'t be foolish'])
"Error: bad argument. Function 'count_char' only accepts strings."
```

Q5.(b) Write a function 'count_words' that takes as parameter 'some_str' as a string and returns a list containing nested_list pairs of the count of each word, ignoring special characters and spaces.

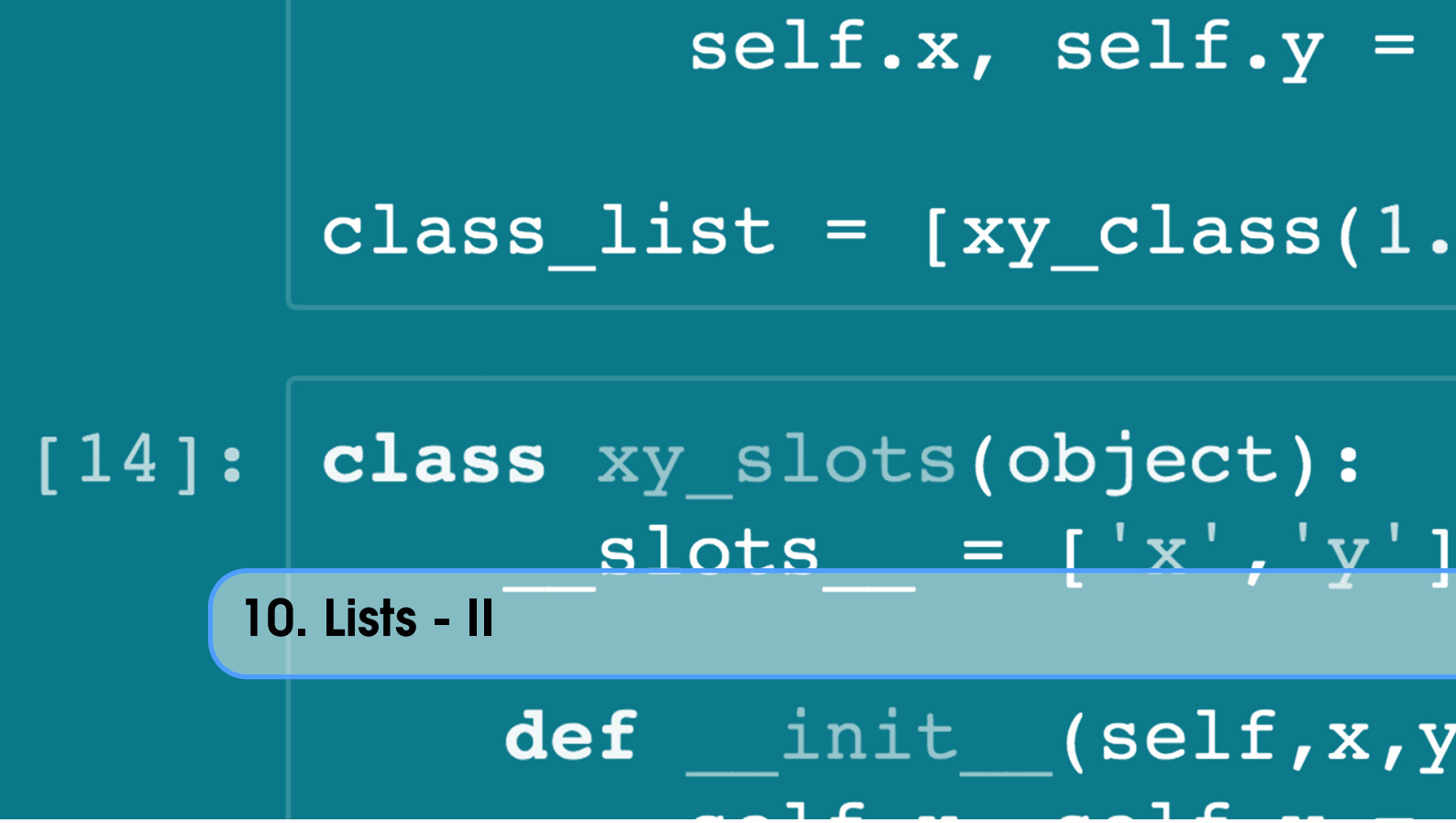
Assume the strings have no contractions i.e. don't, can't, couldn't etc.

- - - - - SAMPLE- - - - -

```
>>> count_words('This might be a bit challenging.')
[['this', 1], ['might', 1], ['be', 1], ['a', 1], ['bit', 1]]
>>> count_words('''If you can manage it, then you are pretty good
at programming my friend, if not, then do not be disheartened, a bit
of determination and hardwork can take you a long way.''')
[['if', 2], ['you', 3], ['can', 2], ['manage', 1], ['it', 1], ['then', 2], ['
are', 1], ['pretty', 1], ['good', 1], ['at', 1], ['programming', 1], ['my', 1
], ['friend', 1], ['not', 2], ['do', 1], ['be', 1], ['disheartened', 1], ['a'
, 2], ['bit', 1], ['of', 1], ['determination', 1], ['and', 1], ['hardwork', 1
], ['take', 1], ['long', 1]]
>>> count_words(['Oh no, not this again'])
"Error: bad argument. Function 'count_words' only accepts strings."
```

X

Lists - II



10. Lists - II

10.1 Searching Lists

The simplest way to search a list is by using a 'for' loop to search it linearly i.e. one element at a time. If the list has nested listed, then their elements can be searched using nested 'for' loops

For example:

```
def in_list(char, lst):
    for nested_lst in lst: # Traversing the nested lists in the main list.
        print(nested_lst)
        for x in nested_lst: # Traversing elements in the nested lists.
            print(x)
            if x == char:
                return True

    return False

lst = [['a','c','d','c'],['n','o','o','r','i'],['z','e','s','t']]
print(in_list('z', lst))
```

Figure 10.1: Searching 101

```
['a', 'c', 'd', 'c']
a c d c
['n', 'o', 'o', 'r', 'i']
n o o r i
['z', 'e', 's', 't']
z True
```

Figure 10.2: Output

10.2 Sorting Lists

Sorting is an important part of data handling. A simple way to sort lists is by comparing each element with all the other elements in the list and swapping their positions with each other as required.

For example:

```
def sort(lst):
    for i in range(len(lst)):
        for j in range(len(lst)):
            if lst[j] >= lst[i]:
                lst[i], lst[j] = lst[j], lst[i]
    return lst
```

Figure 10.3: Sorting

```
>>> sort([2,1,5,23,9])
[1, 2, 5, 9, 23]
```

Figure 10.4: Output

Use this link to see a working visualization of this code and get a better intuition of how sorting works.

10.3 Built-in Functions

Like for strings, Python also has several built-in list functions or methods that can be very useful. A few important ones are listed below. The complete list can be found by executing the following command in the Python/IDLE shell:

```
>>> help(list)
```

Function	Description
lst.index(x)	Returns as an integer the first index where the element 'x' is found or else returns -1
lst.count(x)	Returns as integer the number of occurrences of an element in the list.
lst.insert(x, i)	Inserts element before the i-th index.
lst.pop(i)	Removes the element at the i-th index in the list or by default the element at the last index.
lst.remove(x)	Returns nothing, but removes the first occurrence of the element 'x' in the list.
lst.reverse()	Returns nothing, but reverses the original list.
lst.sort()	Returns nothing, but sorts the original list.

Sample outputs in Python/IDLE shell:

```
>>> a = [2,4,1,3,5,1]
>>> a.index(5)
4
>>> a.count(2)
1
>>> a.insert(6, len(a))
>>> print(a)
[2, 4, 1, 3, 5, 1, 6]
>>> a.pop()
```

```
6
>>> print(a)
[2, 4, 1, 3, 5, 1]
>>> a.pop(4)
5
>>> print(a)
[2, 4, 1, 3, 1]
>>> a.remove(1)
>>> print(a)
[2, 4, 3, 1]
>>> a.reverse()
>>> print(a)
[1, 3, 4, 2]
>>> a.sort()
>>> print(a)
[1, 2, 3, 4]
```

10.4 Strings to Lists

We've already seen how to convert data types from strings to integer or vice versa. It is essential that we know how to convert strings to lists and vice versa. Python has built-in methods to do this.

There are two built-in methods to do this:

1. Using the 'list' function. This creates a list where each element in the list correspondents to a character in the same position. The indexes remain the same.

```
>>> string = 'How do we convert this to a list?'
>>> lst = list(string)
>>> print(lst)
['H', 'o', 'w', ' ', 'd', 'o', ' ', 'w', 'e', ' ', 'c', 'o', 'n', 'v', 'e',
    'r', 't', ' ', 't', 'h', 'i', 's', ' ', 't', 'o', ' ', 'a', ' ', 'l',
    'i', 's', 't', '?']
```

Note: The original string remains the same.

2. Using the 'split' function. By default, if no parameters are passed, then it splits the string apart into list element at empty spaces. If a separator is specified by passing it as a parameter, then the string is split at the separator into list elements.

```
>>> string = ' How do we convert this to a list?'
>>> lst = string.split() # Passing no parameter
>>> print(lst)
['How', 'do', 'we', 'convert', 'this', 'to', 'a', 'list?']
>>>
>>> string = 'How,about,this,?'
>>> lst = string.split(',') # Passing a seperator
>>> print(lst)
['How', 'about', 'this', '?']
```

Splitting is very useful when it comes to file reading and taking multiple user inputs in a single string line. The user maybe instructed to input the required details as comma or space separated values which can then be easily assigned to their respective variables or pass to functions using the split function and slicing.

For example:

```
user_input = 'Ahsan Qadeer M CS 2020'
details = user_input.split()

first_name = details[0]
last_name = details[1]
sex = details[2]
major = details[3]
class_ = details[4]

print(first_name, last_name)
print(sex)
print(major, class_)
```

Figure 10.5: Taking Multiple User Inputs

```
Ahsan Qadeer
M
CS 2020
```

Figure 10.6: Output

For going the other way around: lists to string. We can use the built-in 'join' method. By default, it stitches each element together as a string. However, if a separator is specified, then each element in the string is separator by the separating character.

```
>>> lst = ['j','o','i','n',' ','t','h','i','s']
>>> string = ''.join(lst) # No separator specified
>>> print(string)
'join this'
>>> string = ','.join(lst) # Separator specified
>>> print(string)
'join,this'
```

Note: The original list remains the same and this function only works when all list elements are strings.

10.5 Lab 9 - Exercises

Q1. You are taking a statistics course this semester. The course work extensively uses basic statistical values such as mean, median, and mode. To make your life easier write functions that can do the calculation for you.

(a) The Mean is the average value and is calculated by summing all the data points in a set and dividing them by the number of data points. Write a function 'mean' that takes as parameter 'lst', a list of integers and returns their mean as an integer.

(b) The Median is the middle value in a data set. If there are an odd number of data points, then the median is exactly the middle value, or else it is the mean of the two middle values of an even data set. Write a function 'median' that takes as parameter 'lst', a list of integers and returns their median as an integer.

(c) The Mode is the most reoccurring value in a data set. Write a function named 'mode' that takes as parameter 'lst', a list of numbers and returns the mode value. If there are multiple mode values, then it must return them as a list.

(d) Write a function named 'select' that takes as parameter 'n', an integer that decides which of the three functions needs to be called (1 - Mean, 2 - Median, 3 - Mode). The function also prompts the user to input as a string, space separated values of the data that needs to be analyzed. The data is then passed as a list to function being called. The function should be guarded against invalid parameters.

- - - - - SAMPLE- - - - -

```
>>> select(1)
Data: 1 2 3 4 5
Mean: 3.0
>>> select(2)
Data: 2 1 4 6 8
Median: 4
>>> select(2)
Data: 2 1 4 8 6 5
Median: 6
>>> select(3)
Data: 2 1 4 5 6 4 8 2 0 2
Mode: 2
>>> select(3)
Data: 2 1 4 5 6 6 4 2 0 7
Mode: ['2', '4', '6']
>>> select(10)
'Error: not a valid option. The funcion only accepts the
positive integerss 1, 2 or 3.'
```

Q2. The university needs your help in sorting out its student data. The IT department has generated a list that contains nested lists representing details of every student. However, they are having trouble sort the data in specific order. See if you can help them out by using your exceptional programming skills.

Write a function 'special_sort' that takes as parameter a list 'lst' and returns a sorted version of that list.

The list should be sorted according to the following order of precedence:
1. Class/Batch
2. Major
3. Name (alphabetical order)

- - - - - SAMPLE- - - - -

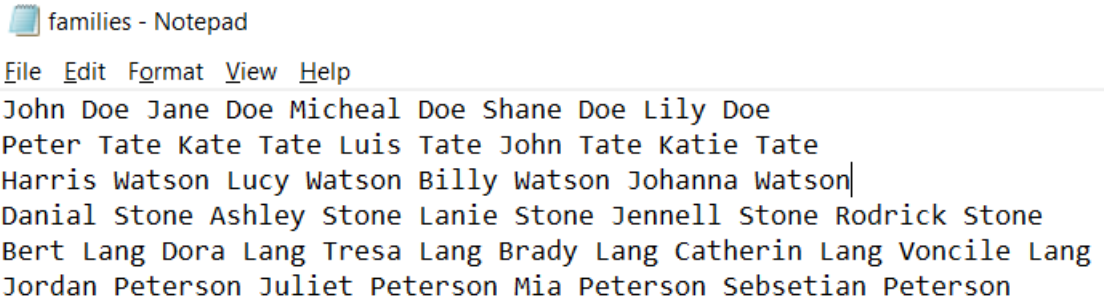
```
>>> special_sort([['Sarwan', 'EE', '2021'],['Luluwalokand Wala', 'CND', '2021'],
['Hamza Junaid', 'EE', '2021'], ['Ahsan Qadeer','CS', '2020'],['Muhammad Ali Bhu
tto', 'EE', '2020'],['Marium Habiby', 'SDP', '2021'],['Adil Ali Khan', 'EE', '20
21']])
[['Ahsan Qadeer', 'CS', '2020'], ['Muhammad Ali Bhutto', 'EE', '2020'], ['Luluwa
lokand Wala', 'CND', '2021'], ['Adil Ali Khan', 'EE', '2021'], ['Hamza Junaid',
'EE', '2021'], ['Sarwan', 'EE', '2021'], ['Marium Habiby', 'SDP', '2021']]
```

Q3.(a) Write a function named 'group_families' that takes as parameter 'filename' as a string and returns a list.

The function reads family details from a file. Each line in the file represents the details of a single family. The first two names are the parents, the rest are their children. Each name consists of a first name and last name.

The function is supposed to extract and arrange each family as a list with two nested lists. One nested list including the parents and the other including their children. Then, each families list will be appended to a main list that contains lists of all the families.

- - - - - SAMPLE- - - - -



```
>>> group_families('families.txt')
[[['John Doe', 'Jane Doe'], ['Micheal Doe', 'Shane Doe', 'Lily Doe']], [
['Peter Tate', 'Kate Tate'], ['Luis Tate', 'John Tate', 'Katie Tate']],
[['Harris Watson', 'Lucy Watson'], ['Billy Watson', 'Johanna Watson']],
[['Danial Stone', 'Ashley Stone'], ['Lanie Stone', 'Jennell Stone', 'Rod
rick Stone']], [['Bert Lang', 'Dora Lang'], ['Tresa Lang', 'Brady Lang',
'Catherin Lang', 'Voncile Lang']], [['Jordan Peterson', 'Juliet Peterson
'], ['Mia Peterson', 'Sebsetian Peterson']]]
```

Q3.(b) Dictionaries are another Python data type/structure. They operate in key value-pairs rather than indexing. Each key has an associated value i.e.

```
>>> dict = {'a key':'a value', 'a':5, 'b':[2,5]}
>>> print(dict['a'])
5
>>> dict['a'] = 10
>>> print(dict['a'])
10
```

Solve part (a) using dictionaries such that family name and its members are key value pairs. Write a function 'group_families_dic' that takes as parameter 'filename' and returns a dictionary.

- - - - - SAMPLE- - - - -

```
>>> group_families_dic('families.txt')
{'Doe': ['John', 'Jane', 'Micheal', 'Shane', 'Lily']
, 'Tate': ['Peter', 'Kate', 'Luis', 'John', 'Katie']
, 'Watson': ['Harris', 'Lucy', 'Billy', 'Johanna'],
'Stone': ['Danial', 'Ashley', 'Lanie', 'Jennell', 'R
odrick'], 'Lang': ['Bert', 'Dora', 'Tresa', 'Brady',
'Catherin', 'Voncile'], 'Peterson': ['Jordan', 'Juli
et', 'Mia', 'Sebsetian']}
```