Introduction to programming for data science

STAT 201

Arvind Krishna

9/20/2022

Table of contents

Preface

This book is currently being written for the course STAT201.

1 Introduction to Jupyter Notebooks and programming in python

This chapter is a very brief introduction to python and Jupyter notebooks. We only discuss the content relevant for applying python to analyze data.

Anaconda: If you are new to python, we recommend downloading the Anaconda installer and following the instructions for installation. Once installed, we'll use the Jupyter Notebook interface to write code.

Quarto: We'll use Quarto to publish the **.ipynb* file containing text, python code, and the output. Download and install Quarto from here.

1.1 Jupyter notebook

1.1.1 Introduction

Jupyter notebook is an interactive platform, where you can write code and text, and make visualizations. You can access Jupyter notebook from the Anaconda Navigator, or directly open the Jupyter Notebook application itself. It should automatically open up in your default browser. The figure below shows a Jupyter Notebook opened with Google Chrome. This page is called the *landing page* of the notebook.

```
<IPython.core.display.Image object>
```

To create a new notebook, click on the `New` button and select the `Python 3` option. You should see a blank notebook as in the figure below.

```
<IPython.core.display.Image object>
```

1.1.2 Writing and executing code

Code cell: By default, a cell is of type *Code*, i.e., for typing code, as seen as the default choice in the dropdown menu below the *Widgets* tab. Try typing a line of python code (say, `2+3`) in an empty code cell and execute it by pressing *Shift+Enter*. This should execute the code, and create an new code cell. Pressing *Ctlr+Enter for Windows (or Cmd+Enter for Mac)* will execute the code without creating a new cell.

Commenting code in a code cell: Comments should be made while writing the code to explain the purpose of the code or a brief explanation of the tasks being performed by the code. A comment can be added in a code cell by preceding it with a # sign. For example, see the comment in the code below.

Writing comments will help other users understand your code. It is also useful for the coder to keep track of the tasks being performed by their code.

```
#This code adds 3 and 5
3+5
```

```
8
```

Markdown cell: Although a comment can be written in a code cell, a code cell cannot be used for writing headings/sub-headings, and is not appropriate for writing lengthy chunks of text. In such cases, change the cell type to *Markdown* from the dropdown menu below the *Widgets* tab. Use any markdown cheat sheet found online, for example, this one to format text in the markdown cells.

Give a name to the notebook by clicking on the text, which says 'Untitled'.

1.1.3 Saving and loading notebooks

Save the notebook by clicking on `File`, and selecting `Save as`, or clicking on the `Save and Checkpoint` icon (below the `File` tab). Your notebook will be saved as a file with an exptension *ipynb*. This file will contain all the code as well as the outputs, and can be loaded and edited by a Jupyter user. To load an existing Jupyter notebook, navigate to the folder of the notebook on the *landing page*, and then click on the file to open it.

1.1.4 Rendering notebook as HTML

We'll use Quarto to print the **.ipynb* file as HTML. Check the procedure for rendering a notebook as HTML here. You have several options to format the file.

You will need to open the command prompt, navigate to the directory containing the file, and use the command: `quarto render filename.ipynb --to html`.

1.2 In-class exercise

1. Create a new notebook.
2. Save the file as `In_class_exercise_1`.
3. Give a heading to the file - `First HTML file`.
4. Print `Today is day 1 of my programming course`.
5. Compute and print the number of seconds in a day.

The HTML file should look like the picture below.

```
<IPython.core.display.Image object>
```

1.3 Python libraries

There are several built-in functions in python like `print()`, `abs()`, `max()`, `sum()` etc., which do not require importing any library. However, these functions will typically be insufficient for a analyzing data. Some of the popular libraries in data science and their primary purposes are as follows:

1. NumPy: Performing numerical operations and efficiently storing numerical data.
2. Pandas: Reading, cleaning and manipulating data.
3. Matplotlib, Seaborn: Visualizing data.
4. SciPy: Performing scientific computing such as solving differential equations, optimization, statistical tests, etc.
5. Scikit-learn: Data pre-processing and machine learning, with a focus on prediction.
6. Statsmodels: Developing statistical models with a focus on inference

A library can be imported using the `import` keyword. For example, a NumPy library can be imported as:

```
import numpy as np
```

Using the `as` keyboard, the NumPy library has been given the name `np`. All the functions and attributes of the library can be called using the *'np.'* prefix. For example, let us generate a sequence of whole numbers upto `10` using the NumPy function arange():

```
np.arange(8)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

Generating random numbers is very useful in python for performing simulations (we'll see in later chapters). The library random is used to generate random numbers such as integers, real numbers based on different probability distributions, etc.

Below is an example of using the `randint()` function of the library for generating random numbers in [a, b], where `a` and `b` are integers.

```
import random as rm
rm.randint(5,10) #This will generate a random number in [5,10]
```

```
7
```

1.4 Debugging and errors

Read sections 1.3 - 1.6 from http://openbookproject.net/thinkcs/python/english3e/way_of_the_program.html

1.5 Terms used in programming

Read section 1.11 from http://openbookproject.net/thinkcs/python/english3e/way_of_the_program.html

2 Variables, expressions and statements

Some of the commonly used objects in python are numbers - integer and float, strings and bool (true/false). The data type of the object can be identified using the in-built python function type(). For example, see the following objects and their types:

```
type(4)
```

```
int
```

```
type(4.4)
```

```
float
```

```
type('4')
```

```
str
```

```
type(True)
```

```
bool
```

2.0.1 In-class exercise

What is the datatype of the following objects?

1. 'This is False'
2. "This is a number"
3. 1000
4. 65.65
5. False

2.1 Constants and Variables

A constant is a value that cannot be changed. It may be a number, string or any other datatype. Below are some examples of printing a constant:

```
print(4)
```

```
4
```

```
print("This is a string and also a constant")
```

```
This is a string and also a constant
```

```
print(False)
```

```
False
```

A variable is an object whose value can be changed. For example, consider the object below:

```
x = 2
```

In the above code the variable `x` has been assigned a value of `2`. However, the value of `x` can be changed:

```
x = 3
print("x =", x)
```

```
x = 3
```

Thus, the object `x` in the above code is a variable that refers to a memory location storing the constant value of 3.

2.1.1 Variable names

There are a some rules for naming variables:

1. A variable name must start with a letter or underscore _
2. A variable name may consist of letters, numbers, and underscores only

For example, some of the valid variable names are *salary, text10, _varname. *Some of the invalid variable names are* salary% 10text, varname)*

3. Variable names are case-sensitive. For example, the variable `Varname` will be different from `varname`.
4. There are certain *reserved words* in python that have some meaning, and cannot be used as variable names. These reserved words are:

```
<IPython.core.display.Image object>
```

Best coding practice: Variables should be named such that they are informative of the value they are storing. For example, suppose we wish to compute the income tax a person has to pay based on their income and tax rate. Below are two ways of naming variables to do this computation:

```
income = 80000
tax_rate = 0.15
print("Income tax = ",income*tax_rate)
```

```
Income tax =  12000.0
```

```
a = 80000
b = 0.15
print("Income tax = ",a*b)
```

```
Income tax =  12000.0
```

The former code chuck is better than the latter one as it makes the code easy to read and understand.

2.1.2 In-class exercise

1. In the statements below, classify the objects as variables or constants:
    1. value = “name”
    2. constant = 7
    3. another_const = “variable”
    4. True_False = True
2. Which of the following variable names are valid:
    1. var.name
    2. var9name
    3. _varname
    4. varname*

2.2 Assignment statements

Values are assigned to variables with the assignment statement (=). An assignment statement may have a constant or an expression on the right hand side of the (=) sign, and a variable name on the left hand side.

For example, the code lines below are assignment statements

```
var = 2
var = var + 3
```

2.3 Expressions

The mathematical operations and their corresponding operators are as follows:

1. Exponent: **
2. Remainder: %
3. Multiplication: *
4. Division: /
5. Addition: +
6. Subtraction: -

The operators above are in decreasing order of precedence, i.e., an exponent will be evaluated before a remainder, a remainder will be evaluated before a multiplication, and so on.

For example, check the precedence of operators in the computation of the following expression:

```
2+3%4*2
```

```
8
```

In case an expression becomes too complicated, use of parenthesis may help clarify the precendence of operators. Parenthesis takes precendence over all the operators listed above. For example, in the expression below, the terms within parenthesis are evaluated first:

```
2+3%(4*2)
```

```
5
```

2.3.1 In-class exercise

Which of the following statements is an assignment statement:

1. x = 5
2. print(x)
3. type(x)
4. x + 4

What will be the result of the following expression:

```
1%2**3*2+1
```

```
3
```

2.4 Converting datatypes

Sometimes a value may have a datatype that is not suitable for using it. For example, consider the variable called *annual_income* in the code below:

```
annual_income = "80000"
```

Suppose we wish to divide *annual_income* by 12 to get the monthly income. We cannot use the variable *monthly_income* directly as its datatype is a string and not a number. Thus, numerical operations cannnot be performed on the variable *annual_income*.

We'll need to convert *annual_income* to an integer. For that we will use the python's in-built `int()` function:

```
annual_income = int(annual_income)
monthly_income = annual_income/12
print("monthly income = ", monthly_income)
```

```
monthly income =  6666.666666666667
```

Similarly, datatypes can be converted from one type to another using in-built python functions as shown below:

```
#Converting integer to string
str(9)
```

```
'9'
```

```
#Converting string to float
float("4.5")
```

```
4.5
```

```
#Converting bool to integer
int(True)
```

```
1
```

Sometimes, conversion of a value may not be possible. For example, it is not possible to convert the variable *greeting* defined below to a number:

```
greeting = "hello"
```

However, in some cases, mathematical operators such as `+` and `*` can be applied on strings. The operator `+` concatenates multiple strings, while the operator `*` can be used to concatenate a string to itself multiple times:

```
"Hi" + " there!"
```

```
'Hi there!'
```

```
"5" + '3'
```

```
'53'
```

```
"5"*8
```

```
'55555555'
```

2.5 User input

Python's in-built input() function can be used to accept an input from the user. For example, suppose we wish the user to onput their age:

```
age = input("Enter your age:")
```

```
Enter your age:34
```

The entered value is stored in the variable `age` and can be used for computation.

2.5.1 In-class exercise

Ask the user to input their year of birth, and print their age.

2.6 Commenting code

The `#` symbol can be used to comment the code. Anything after the `#` sign is ignored by python. Commenting a code may have several purposes, such as:

- Describe what is going to happen in a sequence of code
- Document who wrote the code or other ancillary information
- Turn off a line of code - perhaps temporarily

For example, below is code with a comment to describe the purpose of the code:

```
#Computing number of hours of lecture in this course
print("Total lecture hours of STAT201=",10*3*(5/6))
```

```
Total lecture hours of STAT201= 25.0
```

2.6.1 In-class exercise

Which of the following lines is a comment:

1. #this is a comment
2. ##this may be a comment
3. A comment#

2.7 Programming errors

There are 3 types of errors that can occur in a program - syntax errors, run-time errors, and semantic errors.

2.7.1 Syntax errors

Syntax errors occur if the code is written in a way that it does not comply with the rules / standards / laws of the language (python in this case). For example, suppose a values is assigned to a variable as follows:

```
9value = 2
```

The above code when executed will indicate a syntax error as it violates the rule that a varible name must not start with a number.

2.7.2 Run-time errors

Run-time errors occur when a code is syntactically correct, but there are other issues with the code such as:

- Misspelled or incorrectly capitalized variable and function names
- Attempts to perform operations (such as math operations) on data of the wrong type (ex. attempting to subtract two variables that hold string values)
- Dividing by zero

- Attempts to use a type conversion function such as int on a value that can't be converted to an int

For example, suppose a number is multipled as follws:

```
multiplication_result = x * 4
```

The above code is syntactically correct. However, it will generate an error as the variable `x` has not been defined as a number.

2.7.3 Semantic errors

Semantic errors occur when the code executes without an error being indicated by the compiler. However, it does not work as inteded by the user. For example, consider the following code of mutiplying the number 6 by 3:

```
x = '6'
x * 3
```

```
'666'
```

If it was intended to multiply the number 6, then the variable `x` should have been defined as `x=6` so that `x` has a value of type `integer`. However, in the above code `6` is a `string` type value. When a `string` is multiplied by an integer, say n, it concatenates with itself n times.

2.7.4 In-class exercises:

Identify the type of error from amongst syntax error, semantic error and run-time error

```
income = 2000
tax = .08 * Income
print("tax on", income, "is:", tax)
```

```
income = 2000
tax = .08 x income
print("tax on", income, "is:", tax)
```

```
income = 2000
tax = .08 ** income
print("tax on", income, "is:", tax)
```

2.8 In-class exercise

The formula for computing final amount if one is earning compund interest is given by:

$$A = P\left(1 + \frac{r}{n}\right)^{nt},$$

where:
P = Principal amount (initial investment),
r = annual nominal interest rate,
n = number of times the interest is computed per year,
t = number of years

Write a Python program that assigns the principal amount of \$10000 to variable P, assign to n the value 12, and assign to r the interest rate of 8%. Then have the program prompt the user for the number of years t that the money will be compounded for. Calculate and print the final amount after t years.

What is the amount if the user enters t as 4 years?

```
Number of years:4


25870.703854994306
```

3 Control flow statements

A control flow statement in a computer program determines the individual lines of code to be executed and/or the order in which they will be executed. In this chapter, we'll learn about 3 types of control flow statements:

1. if-elif-else
2. for loop
3. while loop

The first type of control flow satement is `if-elif-else`. This statement helps with conditional execution of code, i.e., the piece of code to be executed is selected based on certain condition(s).

3.0.1 Comparison operators

For testing if conditions are true or false, first we need to learn the opeartors that can be used for comparison. For example, suppose we want to check if two objects are equal, we use the `==` operator:

```
5==6
```

```
False
```

```
x = "hi"
y = "hi"
x==y
```

```
True
```

Below are the python comparison operators and their meanings.

Python code	Meaning
x==y	Produce True if ... x is equal to y
x != y	... x is not equal to y
x > y	... x is greater than y
x < y	... x is less than y
x >= y	... x is greater than or equal to y
x <= y	... x is less than or equal to y

3.0.2 Logical operators

Sometimes we may need to check multiple conditions simultaneously. The logical operator `and` is used to check if all the conditions are true, while the logical operator `or` is used to check if either of the conditons is true.

```
#Checking if both the conditions are true using 'and'
5==5 and 67==68
```

```
False
```

```
#Checking if either condition is true using 'or'
x = 6; y = 90
x<0 or y>50
```

```
True
```

3.0.3 if-elif-else statement

The `if-elif-else` statements can check several conditions, and execute the code corresponding to the condition that is true. Note that there can be as many `elif` statements as required.

Syntax: Python uses indentation to identify the code to be executed if a condition is true. All the code indented within a condition is executed if the condition is true.

Example: Input an integer. Print whether it is positive or negative.

```
number = input("Enter a number:")  #Input an integer
number_integer = int(number)       #Convert the integer to 'int' datatype
if number_integer>0:               #Check if the integer is positive
    print("Number is positive")
```

```
else:
    print("Number is negative")
```

```
Enter a number:-9
Number is negative
```

In the above code, note that anything entered by the user is taken as a string datatype by python. However, a string cannot be positive or negative. So, we converted the number input by the user to integer to check if it was positive or negative.

There may be multiple statements to be executed if a condition is true. See the example below.

Example: Input a number. Print whether it is positive, negative or zero. If it is negative, print its absolute value.

```
number = input("Enter a number:")
number_integer = int(number)
if number_integer>0:
    print("Number is positive")
elif number_integer==0:
    print("Number is zero")
else:
    print("Number is negative")
    print("Absolute value of number = ", abs(number_integer))
```

```
Enter a number:0
Number is zero
```

3.0.4 Practice exercise 1

Input a number. Print whether its odd or even.

Solution:

```
num = int(input("Enter a number: "))
if num%2==0:           #Checking if the number is divisible by 2
    print("Number is even")
else:
    print("Number is odd")
```

```
Enter a number: 5
Number is odd
```

3.0.5 Try-except

If we suspect that some lines of code may produce an error, we can put them in a `try` block, and if an error does occur, we can use the `except` block to instead execute an alternative piece of code. This way the program will not stop if an error occurs within the `try` block, and instead will be directed to execute the code within the `except` block.

Example: Input an integer from the user. If the user inputs a valid integer, print whether it is a multiple of 3. However, if the user does not input a valid integer, print a message saying that the input is invalid.

```
num = input("Enter an integer:")

#The code lines within the 'try' block will execute as long as they run without error
try:
    num_int = int(num)
    if num_int%3==0:
        print("Number is a multiple of 3")
    else:
        print("Number is not a multiple of 3")

#The code lines within the 'except' block will execute only if the code lines within the '
except:
    print("Input must be an integer")
```

```
Enter an integer:hi
Input must be an integer
```

3.0.6 Practice exercise 2

3.0.6.1

Ask the user to enter their exam score. Print the grade based on their score as follows:

Score	Grade
(90,100]	A
(80,90]	B

Score	Grade
[0,80]	C

If the user inputs a score which is not a number in [0,100], print invalid entry.

Solution:

```
score = input("Enter exam score:")
try:
    score_num = float(score)
    if score_num>90 and score_num<=100:
        print("Grade: A")
    elif score_num>80 and score_num<=90:
        print("Grade: B")
    elif score_num>=0 and score_num<=80:
        print("Grade: C")
    else:
        print("Invalid score")     #If a number is less than 0 or more than 100
except:
    print("Invalid input")         #If the input is not a number
```

```
Enter exam score:90
Grade: B
```

3.0.6.2

Nested if-elif-else statements: This question will lead you to create nested `if` statements, i.e., an `if` statement within another `if` statement.

Think of a number in [1,5]. Ask the user to guess the number.

- If the user guesses the number correctly, print "Correct in the first attempt!", and stop the program. Otherwise, print "Incorrect! Try again" and give them another chance to guess the number.
- If the user guesses the number correctly in the second attempt, print "Correct in the second attempt", otherwise print "Incorrect in both the attempts, the correct number is:", and print the correct number.

3.1 Loops

With loops, a piece of code can be executed repeatedly for a fixed number of times or until a condition is satistified.

3.1.1 `for` loop

With a `for` loop, a piece of code is executed a fixed number of times.

We typically use `for` loops with an in-built python function called `range()` that supports `for` loops. Below is its description.

range(): The range() function returns a sequence of evenly-spaced integer values. It is commonly used in `for` loops to define the sequence of elements over which the iterations are performed.

Below is an example where the `range()` function is used to create a sequence of whole numbers upto 10. Ignore the `list()` function in the code below, as it will be introduced later.

```
print(list(range(10)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Note that the last element is one less than the integer specified in the `range()` function.

Using the `range()` function, the `for` loop can iterate over a sequence of numbers. See the example below.

Example: Print the first n elements of the Fibonacci sequence, where n is an integer input by the user, such that $n>2$. In a fibonacci sequence, each number is the sum of the preceding two numbers, and the sequence starts from 0,1. The sequence is as follows:

0,1,1,2,3,5,8,13,....

```
n=int(input("Enter number of elements:"))
n1=0;n2=1
print(n1)
print(n2)
for i in range(n-2):  #Since two numbers of the sequence are already printed, n-2 numbers
    n3 = n1+n2
    print(n3)
    n1 = n2
    n2 = n3
```

```
print("These are the first", n, "elements of the fibonacci series")
```

```
Enter number of elements:6
0
1
1
2
3
5
These are the first 6 elements of the fibonacci series
```

As in the `if-elif-else` statement, the `for` loop uses identation to indicate the piece of code to be run repeatedly.

Note that we have used an in-built python function

3.1.2 `while` loop

With a `while` loops, a piece of code is executed repeatedly until certain condition(s) hold.

Example: Print all the elements of the Fibonacci sequence less than n, where n is an integer input by the user, such that $n>2$. In a fibonacci sequence, each number is the sum of the preceding two numbers, and the sequence starts from 0,1. The sequence is as follows:

0,1,1,2,3,5,8,13,….

```
n=int(input("Enter the value of n:"))
n1=0;n2=1
print(n1)
while n2<n:
    print(n2)
    n3 = n1+n2
    n1 = n2
    n2 = n3
print("These are all the elements of the fibonacci series less than", n)
```

```
Enter the value of n:50
0
1
1
2
```

```
3
5
8
13
21
34
These are all the elements of the fibonacci series less than 50
```

3.1.3 Practice exercise 3

Print the prime numbers starting from 2, and less than n where n is a positive integer input by the user.

4 Data structures

In this chapter we'll learn about the python data structures that are often used or appear while analyzing data.

Tuple is a sequence of python objects, with two key characeterisics: (1) the number of objects are fixed, and (2) the objects are immutable, i.e., they cannot be changed.

Tuple can be defined as a sequence of python objects separated by commas, and enclosed in rounded brackets (). For example, below is a tuple containing three integers.

```
tuple_example = (2,7,4)
```

We can check the data type of a python object using the *type()* function. Let us check the data type of the object *tuple_example*.

```
type(tuple_example)
```

```
tuple
```

Elements of a tuple can be extracted using their index within square brackets. For example the second element of the tuple *tuple_example* can be extracted as follows:

```
tuple_example[1]
```

```
7
```

Note that an object of a tuple cannot be modified. For example, consider the following attempt in changing the second element of the tuple *tuple_example*.

```
tuple_example[1] = 8
```

```
TypeError: 'tuple' object does not support item assignment
```

The above code results in an error as tuple elements cannot be modified.

4.0.1 Concatenating tuples

Tuples can be concatenated using the + operator to produce a longer tuple:

```
(2,7,4) + ("another", "tuple") + ("mixed","datatypes",5)
```

```
(2, 7, 4, 'another', 'tuple', 'mixed', 'datatypes', 5)
```

Multiplying a tuple by an integer results in repeitition of the tuple:

```
(2,7,"hi") * 3
```

```
(2, 7, 'hi', 2, 7, 'hi', 2, 7, 'hi')
```

4.0.2 Unpacking tuples

If tuples are assigned to an expression containing multiple variables, the tuple will be unpacked and each variable will be assigned a value as per the order in which is appears. See the example below.

```
x,y,z  = (4.5, "this is a string", (("Nested tuple",5)))
```

```
x
```

```
4.5
```

```
y
```

```
'this is a string'
```

```
z
```

```
('Nested tuple', 5)
```

If we are interested in retrieving only some values of the tuple, the expression *_ can be used to discard the other values. Let's say we are interested in retrieving only the first and the last two values of the tuple:

```
x,*_,y,z  = (4.5, "this is a string", (("Nested tuple",5)),"99",99)
```

```
x
```

```
4.5
```

```
y
```

```
'99'
```

```
z
```

```
99
```

4.0.3 Tuple methods

A couple of useful tuple methods are `count`, which counts the occurences of an element in the tuple and `index`, which returns the position of the first occurance of an element in the tuple:

```
tuple_example = (2,5,64,7,2,2)
```

```
tuple_example.count(2)
```

```
3
```

```
tuple_example.index(2)
```

```
0
```

Now that we have an idea about tuple, let us try to think where it can be used.

```
<IPython.core.display.HTML object>
```

4.1 List

List is a sequence of python objects, with two key characeterisics that differentiate them from tuples: (1) the number of objects are variable, i.e., objects can be added or removed from a list, and (2) the objects are mutable, i.e., they can be changed.

List can be defined as a sequence of python objects separated by commas, and enclosed in square brackets []. For example, below is a list containing three integers.

```
list_example = [2,7,4]
```

4.1.1 Adding and removing elements in a list

We can add elements at the end of the list using the *append* method. For example, we append the string 'red' to the list *list_example* below.

```
list_example.append('red')
```

```
list_example
```

```
[2, 7, 4, 'red']
```

Note that the objects of a list or tuple can be of different datatypes.

An element can be added at a specific location of the list using the *insert* method. For example, if we wish to insert the number 2.32 as the second element of the list *list_example*, we can do it as follows:

```
list_example.insert(1,2.32)
```

```
list_example
```

```
[2, 2.32, 7, 4, 'red']
```

For removing an element from the list, the *pop* and *remove* methods may be used. The *pop* method removes an element at particular index, while the *remove* method removes the element's first occurence in the list by its value. See the examples below.

Let us say, we need to remove the third element of the list.

```
list_example.pop(2)
```

```
7
```

```
list_example
```

```
[2, 2.32, 4, 'red']
```

Let us say, we need to remove the element 'red'.

```
list_example.remove('red')
```

```
list_example
```

```
[2, 2.32, 4]
```

```
#If there are multiple occurences of an element in the list, the first occurence will be r
list_example2 = [2,3,2,4,4]
list_example2.remove(2)
list_example2
```

```
[3, 2, 4, 4]
```

For removing multiple elements in a list, either `pop` or `remove` can be used in a for loop, or a for loop can be used with a condition. See the examples below.

Let's say we need to remove intergers less than 100 from the following list.

```
list_example3 = list(range(95,106))
list_example3
```

```
[95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105]
```

```
#Method 1: For loop with remove
list_example3_filtered = list(list_example3) #
```

```
for element in list_example3:
    #print(element)
    if element<100:
        list_example3_filtered.remove(element)
print(list_example3_filtered)
```

```
[100, 101, 102, 103, 104, 105]
```

Q: What's the need to define a new variable 'list_example3_filtered' in the above code?

Replace list_example3_filtered with list_example3 and identify the issue.

```
#Method 2: For loop with condition
[element for element in list_example3 if element>100]
```

```
[101, 102, 103, 104, 105]
```

4.1.2 Concatenating lists

As in tuples, lists can be concatenated using the + operator:

```
import time as tm
```

```
list_example4 = [5,'hi',4]
list_example4 = list_example4 + [None,'7',9]
list_example4
```

```
[5, 'hi', 4, None, '7', 9]
```

For adding elements to a list, the `extend` method is preferred over the + operator. This is because using the + operator creates a new list, while the `extend` method adds elements to an existing list.

```
list_example4 = [5,'hi',4]
list_example4.extend([None, '7', 9])
list_example4
```

```
[5, 'hi', 4, None, '7', 9]
```

4.1.3 Sorting a list

A list can be sorted using the `sort` method:

```
list_example5 = [6,78,9]
list_example5.sort(reverse=True) #the reverse argument is used to specify if the sorting i
list_example5
```

```
[78, 9, 6]
```

4.1.4 Slicing a list

We may extract or update a section of the list by passing the starting index (say `start`) and the stopping index (say `stop`) as `start:stop` to the index operator []. This is called *slicing* a list. For example, see the following example.

```
list_example6 = [4,7,3,5,7,1,5,87,5]
```

Let us extract a slice containing all the elements starting from the the 3rd position upto the 7th position.

```
list_example6[2:7]
```

```
[3, 5, 7, 1, 5]
```

Note that while the element at the `start` index is included, the element with the `stop` index is excluded in the above slice.

If either the `start` or `stop` index is not mentioned, the slicing will be done from the beginning or utop the end of the list, respectively.

```
list_example6[:7]
```

```
[4, 7, 3, 5, 7, 1, 5]
```

```
list_example6[2:]
```

```
[3, 5, 7, 1, 5, 87, 5]
```

To slice the list relative to the end, we can use negative indices:

```
list_example6[-4:]
```

```
[1, 5, 87, 5]
```

```
list_example6[-4:-2:]
```

```
[1, 5]
```

An extra colon (':') can be used to slice every ith element of a list.

```
#Selecting every 3rd element of a list
list_example6[::3]
```

```
[4, 5, 5]
```

```
#Selecting every 3rd element of a list from the end
list_example6[::-3]
```

```
[5, 1, 3]
```

```
#Selecting every element of a list from the end or reversing a list
list_example6[::-1]
```

```
[5, 87, 5, 1, 7, 5, 3, 7, 4]
```

Now that we have an idea about lists, let us try to think where it can be used.

```
<IPython.core.display.HTML object>
```

Now that we have learned about lists and tuples, let us compare them.

Q: A list seems to be much more flexible than tuple, and can replace a tuple almost everywhere. Then why us

A: The additional flexibility of a list comes at the cost of efficiency. Some of the advatages of a tuple over a list are as follows:

1. Since a list can be extended, space is over-allocated when creating a list. A tuple takes less storage space as compared to a list of the same length.
2. Tuples are not copied. If a tuple is assigned to another tuple, both tuples point to the same memory location. However, if a list is assigned to another list, a new list is created consuming the same memory space as the orignial list.
3. Tuples refer to their element directly, while in a list, there is an extra layer of pointers that refers to their elements. Thus it is faster to retrieve elements from a tuple.

The examples below illustrate the above advantages of a tuple.

```
#Example showing tuples take less storage space than lists for the same elements
tuple_ex = (1, 2, 'Obama')
list_ex = [1, 2, 'Obama']
print("Space taken by tuple =",tuple_ex.__sizeof__()," bytes")
print("Space taken by list =",list_ex.__sizeof__()," bytes")
```

```
Space taken by tuple = 48  bytes
Space taken by list = 64  bytes
```

```
#Examples showing that a tuples are not copied, while lists can be copied
tuple_copy = tuple(tuple_ex)
print("Is tuple_copy same as tuple_ex?", tuple_ex is tuple_copy)
list_copy = list(list_ex)
print("Is list_copy same as list_ex?",list_ex is list_copy)
```

```
Is tuple_copy same as tuple_ex? True
Is list_copy same as list_ex? False
```

```
#Examples showing tuples takes lesser time to retrieve elements
import time as tm
tt = tm.time()
list_ex = list(range(1000000)) #List containinig whole numbers upto 1 million
a=(list_ex[::-2])
print("Time take to retrieve every 2nd element from a list = ", tm.time()-tt)

tt = tm.time()
tuple_ex = tuple(range(1000000)) #tuple containinig whole numbers upto 1 million
a=(tuple_ex[::-2])
print("Time take to retrieve every 2nd element from a tuple = ", tm.time()-tt)
```

```
Time take to retrieve every 2nd element from a list =  0.03579902648925781
Time take to retrieve every 2nd element from a tuple =  0.02684164047241211
```

4.2 Dictionary

A dictionary consists of key-value pairs, where the keys and values are python objects. While values can be any python object, keys need to be immutable python objects, like strings, intergers, tuples, etc. Thus, a list can be a value, but not a key, as a elements of list can be changed. A dictionary is defined using the keyword `dict` along with curly braces, colons to separate keys and values, and commas to separate elements of a dictionary:

```
dict_example = {'USA':'Joe Biden', 'India':'Narendra Modi', 'China':'Xi Jinping'}
```

Elements of a dictionary can be retrieved by using the corresponding key.

```
dict_example['India']
```

```
'Narendra Modi'
```

4.2.1 Adding and removing elements in a dictionary

New elements can be added to a dictionary by defining a key in square brackets and assiging it to a value:

```
dict_example['Japan'] = 'Fumio Kishida'
dict_example['Countries'] = 4
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Japan': 'Fumio Kishida',
 'Countries': 4}
```

Elements can be removed from the dictionary using the `del` method or the `pop` method:

```
#Removing the element having key as 'Countries'
del dict_example['Countries']
```

```
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Japan': 'Fumio Kishida'}
```

```
#Removing the element having key as 'USA'
dict_example.pop('USA')
```

```
'Joe Biden'
```

```
dict_example
```

```
{'India': 'Narendra Modi', 'China': 'Xi Jinping', 'Japan': 'Fumio Kishida'}
```

New elements can be added, and values of exisiting keys can be changed using the `update` method:

```
dict_example = {'USA':'Joe Biden', 'India':'Narendra Modi', 'China':'Xi Jinping','Countrie
dict_example
```

```
{'USA': ['Joe Biden'],
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Countries': 3}
```

```
dict_example.update({'Countries':4, 'Japan':'Fumio Kishida'})
```

```
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Countries': 4,
 'Japan': 'Fumio Kishida'}
```

4.3 Functions

If an algorithm or block of code is being used several times in a code, then it can be separatey defined as a function. This makes the code more organized and readable. For example, let us define a function that prints prime numbers between `a` and `b`, and returns the number of prime numbers found.

```
#Function definition
def prime_numbers (a,b=100):
    num_prime_nos = 0

    #Iterating over all numbers between a and b
    for i in range(a,b):
        num_divisors=0

        #Checking if the ith number has any factors
        for j in range(2, i):
            if i%j == 0:
                num_divisors=1;break;

        #If there are no factors, then printing and counting the number as prime
        if num_divisors==0:
            print(i)
            num_prime_nos = num_prime_nos+1

    #Return count of the number of prime numbers
    return num_prime_nos
```

In the above function, the keyword `def` is used to define the function, `prime_numbers` is the name of the function, `a` and `b` are the arguments that the function uses to compute the output.

Let us use the defined function to print and count the prime numbers between 40 and 60.

```
#Printing prime numbers between 40 and 60
num_prime_nos_found = prime_numbers(40,60)
```

```
41
43
47
53
59
```

```
num_prime_nos_found
```

```
5
```

If the user calls the function without specifying the value of the argument `b`, then it will take the default value of `100`, as mentioned in the function definition. However, for the argument `a`, the user will need to specify a value, as their is no value defined as a default value in the function definition.

4.3.1 Global and local variables with respect to a function

A variable defined within a function is local to that function, while a variable defined outside the function is global with respect to that function. In case a variable with the same name is defined both outside and inside a function, it will refer to its global or local value, depending on where it occurs.

The example below shows a variable with the name `var` referring to its local value when called within the function, and global value when called outside the function.

```
var = 5
def sample_function(var):
    print("Local value of 'var' within 'sample_function()'= ",var)

sample_function(4)
print("Global value of 'var' outside 'sample_function()' = ",var)
```

```
Local value of 'var' within 'sample_function()'=  4
Global value of 'var' outside 'sample_function()' =  5
```

5 Functions

As the words suggests, *functions* are a piece of code that have a specific function or purpose. As an analogy, if a human is a computer program, then the mind can be considered to be a function, which has purpose of thinking, eyes can be another function, which have a purpose of seeing. These functions are called upon by the human when needed.

Similarly, in case of a computer program, functions are a piece of code, that perform a specific task, when called upon by the program. Instead of being defined as a function, the piece of code can also be used directly whenever it is needed in a program. However, defining a frequently-used piece of code as a function has the following benefits:

1. It reduces the number of lines of code, as the lines of code need to be written just once in the function definition. Thereafter, the function is called by its name, wherever needed in the program. This makes the code compact, and enhances readibility.
2. It makes the process of writing code easier, as the user needs to just type the name of the function, wherever it is needed, instead of pasting lines of code.
3. It can be used in different programs, thereby saving time in writing other programs.

To put it more formally, a function is a piece of code that takes arguments (if any) as input, performs computations or tasks, and then returns a result or results.

5.1 Defining a function

Look at the function defined below. It asks the user to input a number, and prints whether the number is odd or even.

```
#This is an example of a function definition

#A function definition begins with the 'def' keyword followed by the name of the function.
#Note that 'odd_even()' is the name of the function below.
def odd_even():
    num = int(input("Enter an integer:"))
    if num%2==0:
        print("Even")
```

```
    else:
        print("Odd")   #Function definition ends here

print("This line is not a part of the function as it is not indented") #This line is not a
```

```
This line is not a part of the function as it is not indented
```

Note that the function is defined using the `def` keyword. All the lines within the function definition are indented. The indentation shows the lines of code that below to the function. When the indentation stops, the function definition is considered to have ended.

Whenever the user wishes to input a number and print whether it is odd or even, they can call the function defined above by its name as follows:

```
odd_even()
```

```
Enter an integer:5
Odd
```

5.2 Arguments of a function

Note that the function defined above needs no input when called. However, sometimes we may wish to define a function that takes input(s), and performs computations on the inputs to produce an output. These input(s) are called argument(s) of a function.

5.2.1 Function with an argument

Let us change the previous example to write a function that takes an integer as an input argument, and prints whether it is odd or even:

```
#This is an example of a function definition that has an argument
def odd_even(num):
    if num%2==0:
        print("Even")
    else:
        print("Odd")
```

We can use the function whenever we wish to find a number is odd or even. For example, if we wish to find that a number input by the user is odd or even, we can call the function with the user input as its argument.

```
number = int(input("Enter an integer:"))
odd_even(number)
```

```
Enter an integer:6
Even
```

Note that the above function needs an argument as per the function definition. It will produce an error if called without an argument:

```
odd_even()
```

```
TypeError: odd_even() missing 1 required positional argument: 'num'
```

5.2.2 Function with an argument having a default value

To avoid errors as above, sometimes is a good idea to assign a default value to the argument in the function definition:

```
#This is an example of a function definition that has an argument with a default value
def odd_even(num=0):
    if num%2==0:
        print("Even")
    else:
        print("Odd")
```

Now, we can call the function without an argument. The function will use the default value of the argument specified in the function definition.

```
odd_even()
```

```
Even
```

5.2.3 Function with multiple arguments

A function can have as many arguments as needed. Multiple arguments are separated by commas. For example, below is a function that inputs two strings, concatenates them with a space in between, and prints the output:

```
def concat_string(string1, string2):
    print(string1+' '+string2)
```

```
concat_string("Hi", "there")
```

```
Hi there
```

5.3 Functions that return objects

Until now, we saw functions

References