Introduction to programming for data science

STAT 201

Arvind Krishna

9/20/22

Table of contents

Preface

This book is currently being written for the course STAT201.

Part I

Python

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 Practice exercise 1

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 Practice exercise 2

2.1.2.1 Variables or constants?

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

2.1.2.2 Valid variable names?

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 Practice exercise 3

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
```

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 Practice exercise 4

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 Practice exercise 5

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 variable 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 Practice exercise 6

Suppose we wish to compute tax using the income and the tax rate. Identify the type of error from amongst syntax error, semantic error and run-time error in the following pieces of code.

```
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 Practice exercise 7

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?

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 operators 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.

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:
    #Converting the input to integer, as user input is a string
    num_int = int(num)

    #checking if the integer is a multiple of 3
    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
[0,80]	C

If the user inputs a score which is not a number, print "*Exam score must be a number*". If the input is a number, but not in [0,100], print "*Exam score must be in [0, 100]*".

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:"))

#Initializing the sequence to start from 0, 1
n1=0;n2=1

#Printing the first two numbers of the sequence
print(n1)
print(n2)

for i in range(n-2):  #Since two numbers of the sequence are already printed, n-2 numbers

    #Computing the next number of the sequence as the summation of the previous two number
    n3 = n1+n2
    print(n3)

    #As 'n3' is already printed, it is no longer the next number of the sequence.
    #Thus, we move the values of the variables n1 and n2 one place to the right to compute
    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:"))

#Initializing the sequence to start from 0, 1
n1=0;n2=1

#Printing the first number of the sequence
print(n1)

while n2<n:

    #Print the next number of the sequence
    print(n2)

    #Comptuing the next number of the sequence as the summation of the previous two number
    n3 = n1+n2

    #As n2 is already printed, assigning n2 to n3, so that the next number of the sequence
    #Assigning n1 to n2 as n1 has already been used to compute the next number of the seqe
    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

3.1.3.1

Write a program that identifies whether a number input by the user is prime or not.

3.1.3.2

Update the program above to print the prime numbers starting from 2, and less than n where n is a positive integer input by the user.

3.2 `break` statement

The break statement is used to unconditionally exit the innermost loop.

For example, suppose we need to keep asking the user to input year of birth and compute the corresponding age, until the user enters 1900 as the year of birth.

```
#The loop will continue to run indefinitely as the condition 'True' is always true
while True:
    year = int(input("Enter year of birth:"))
    if year==1900:
        break        #If the user inputs 1900, then break out of the loop
    else:
        print("Age = ",2022-year)    #Otherwise compute and print the age
```

```
Enter year of birth:1987
Age =  35
Enter year of birth:1995
Age =  27
Enter year of birth:2001
Age =  21
Enter year of birth:1900
```

3.2.1 Practice exercise 4

Write a program that finds and prints the largest factor of a number input by the user. Check the output if the user inputs 133.

3.3 `continue` statement

The continue statement is used to continue with the next iteration of the loop without executing the lines of code below it.

For example, consider the following code:

```
for i in range(10):
    if i%2==0:
        continue
    print(i)
```

```
1
3
5
7
9
```

When the control flow reads the statement `continue`, it goes back to the beginning of the `for` loop, and ignores the lines of code below the statement.

3.3.1 Practice exercise 5:

Write a program that asks the user the question, "How many stars are in the Milky Way (in billions)?". If the user answers 100, the program should print correct, and stop. However, if the user answers incorrectly, the program should print "incorrect", and ask them if they want to try again. The program should continue to run until the user answers correctly, or they want to stop trying.

3.4 Loops with strings

Loops can be used to iterate over a string, just like we used them to iterate over a sequence of integers.

Consider the following string:

```
sentence = "She sells sea shells on the sea shore"
```

The i^{th} character of the string can be retrieved by its index. For example, the first character of the string `sentence` is:

```
sentence[0]
```

```
'S'
```

Slicing a string:

A part of the string can be sliced 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 string. For a string `S`, the characters starting from the index `start` upto the index `stop`, but not including `stop`, can be sliced as `S[start:stop]`.

For example, the slice of the string `sentence` from index `4` to index `9`, but not including `9` is:

```
sentence[4:9]
```

```
'sells'
```

Example:

Input a string, and count and print the number of "t"s.

```
string = input("Enter a sentence:")

#Initializing a variable 'count_t' which will store the number of 't's in the string
count_t = 0

#Iterating over the entire length of the string.
#The length of the string is given by the len() function
for i in range(len(string)):
```

```
    #If the ith character of the string is 't', then we count it
    if string[i]=='t':
        count_t = count_t+1

print("Number of 't's in the string = ",count_t)
```

```
Enter a sentence:Getting a tatto is not a nice experience
Number of 't's in the string =  6
```

3.4.1 Practice exercise 6

Write a program that asks the user to input a string, and print the number of "*the*"s in the string.

4 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 readability.
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.

4.1 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
```

4.2 Parameters and 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 parameter(s) of a function. When a function is called, the value(s) of these parameter(s) must be specified as argument(s) to the function.

4.2.1 Function with a parameter

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'
```

4.2.2 Function with a parameter having a default value

To avoid errors as above, sometimes is a good idea to assign a default value to the parameter 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 parameter specified in the function definition.

```
odd_even()
```

```
Even
```

4.2.3 Function with multiple parameters

A function can have as many parameters as needed. Multiple parameters/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
```

4.2.4 Practice exercise 1

Write a function that prints prime numbers between two real numbers - `a` and `b`, where `a` and `b` are the parameters of the function. Call the function and check the output with a = 60, b = 80.

4.3 Functions that return objects

Until now, we saw functions that print text. However, the functions did not `return` any object. For example, the function `odd_even` prints whether the number is odd or even. However, we did not save this information. In future, we may need to use the information that whether the number was odd or even. Thus, typically, we return an object from the function definition, which consists of the information we may need in the future.

The example `odd_even` can be updated to return the text "odd" or "even" as shown below:

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

The function above returns a string "Odd" or "Even", depending on whether the number is odd or even. This result can be stored in a variable, which can be used later.

```
response=odd_even(3)
response
```

```
'Odd'
```

The variable `response` now refers to the object where the string "Odd" or "Even" is stored. Thus, the result of the computation is stored, and the variable can be used later on in the program. Note that the control flow exits the function as soon as the first `return` statement is executed.

Figure 4.1 below shows the terminology associated with functions.

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

Figure 4.1: Terminology associated with functions

4.4 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 value outside the function and local value within the function.

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
```

4.5 Built-in python functions

So far we have seen user-defined functions in this chapter. These functions were defined by us, and are not stored permanently in the python compiler. However, there are some functions that come built-in with python and we can use them directly without defining them. These built-in functions can be see here. For example the built-in function `max()` computes the max of numeric values:

```
max(1,2,3)
```

```
3
```

Another example is the `round()` function that rounds up floating point numbers:

```
round(3.7)
```

```
4
```

4.6 Python libraries

Other than the built-in functions, python has hundreds of thousands of libraries that contain several useful functions. These libraries are contributed by people around the world as python is an open-source platform. Some of the libraries popular in data science, and their purposes are the following:

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
```

4.6.1 Practice exercise 2

Generate a random number between [-5,5]. Do this 10,000 times. Find the mean of all the 10,000 random numbers generated.

A Assignment 1

1. You may talk to a friend, discuss the questions and potential directions for solving them. However, you need to write your own solutions and code separately, and not as a group activity.
2. Do not write your name on the assignment.
3. Write your code in the *Code* cells of the Jupyter notebook. Ensure that the solution is written neatly enough to understand and grade.
4. Use Quarto to print the *.ipynb* file as HTML. 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`. Submit the HTML file.
5. There are 5 points for clealiness and organization. The breakdow is as follows:

- Must be an HTML file rendered using Quarto (1.5 pts).
- There aren't excessively long outputs of extraneous information (e.g. no printouts of unnecessary results without good reason, there aren't long printouts of which iteration a loop is on, there aren't long sections of commented-out code, etc.) (1 pt)
- There is no piece of unnecessary / redundant code, and no unnecessary / redundant text (1 pt)
- The code should be commented and clearly written with intuitive variable names. For example, use variable names such as number_input, factor, hours, instead of a,b,xyz, etc. (1.5 pts)

6. The assignment is worth 100 points, and is due on **13th April 2023 at 11:59 pm**.

A.1 Computation

You look at the clock and it is exactly 2pm. You set an alarm to go off in 510 hours. At what time does the alarm go off? If the answer is say, 4 pm, then your code should print - `"The alarm goes off at 4 pm"`.

(2 points)

A.2 Generalizing computation

Write a program to solve the general version of the above problem. Ask the user for - (1) the time now (in hours), and (2) the number of hours for the alarm to go off. Your program should output the time at which the alarm goes off. Both the user inputs must be in {0, 1, 2..., 22, 23}. If the answer is, say 14:00 hours, then your program should print - "The alarm goes off at 14:00 hours.

Show the output of your program when the user inputs `7` as the current time, and `95` as the number of hours for the alarm to go off.

(4 points)

A.3 Finding prime factors

A.3.1 Prime or not

Write a program that checks if a positive integer is prime or not. Show the output when the program is used to check if `89` is prime or not.

(2 points)

A.3.2 Factors

Prompt the user to input a positive integer. Write a program that prints the **factors** of the positive integer input by the user. Show the output of the program if the user inputs `190`.

(2 points)

A.3.3 Prime factors

Prompt the user to input a positive integer. Update the program in 2(b) to print the **prime factors** of the positive integer input by the user. Show the output of the program if the user inputs `190`.

(8 points)

A.3.4 Adding robustness to the program

Update the program in 2(c), so that it prints "Incorrect input, please enter positive integer" if the user does not enter a positive integer, and then prompts the user to input a positive integer. The program should continue to prompt the user to enter a positive integer until the user successfully enters a positive integer. Show the output of the program if the user enters `"seventy"` in the first attempt, `"#70"` in the second attempt, and `70` in the third attempt.

(12 points)

A.4 Number of words

Prompt the user to input an english sentence. Write a program that counts and prints the number of words in the sentence input by the user. The program should continue to run until the user inputs the sentence - "end program". Show the output of the program if the user enters `"this is the time to sleep"` in the first attempt, `"this is too much work for a day"` in the second attempt, and `"end program"` in the third attempt.

Hint: Count the number of spaces

(10 points)

A.5

A.6 Survival of rabbits

In many environments, two or more species compete for the available resources. Classic predator–prey equations have been used to simulate or predict the dynamics of biological systems in which two species interact, one as a predator and the other as prey. You will use a simplified version of the Lotka-Volterra equations for modeling fox/rabbit populations, described below.

Let the following variables be defined as:

r_t: The number of prey (rabbits) at time t, where t corresponds to a certain year.

f_t: The number of predators (foxes) at time t, where t corresponds to a certain year.

α: The birth rate of prey.

β: The death rate of prey (depends on predator population).

γ: The birth rate of predators (depends on prey population).

δ: The death rate of predators.

Then, we can define the populations of the next time period or the next year (t+1) using the following system of equations:

$$r_{t+1} = r_t + \alpha r_t - \beta r_t f_t,$$

$$f_{t+1} = f_t + \gamma f_t r_t - \delta f_t$$

A.6.1 Number of rabbits and foxes

Write a program that uses the following parameter values, and calculates and prints the populations of the rabbits and foxes for each year upto the next 14 years. Since the number of rabbits and foxes cannot be floating-point numbers, use the in-built python function `round()` to round-off the calculated values to integers. Also, we cannot have negative rabbits or negative foxes, so if the population values are ever negative, consider the population to be zero instead.

$r_0 = 500$

$f_0 = 1$

$\alpha = 0.2$

$\beta = 0.005$

$\gamma = 0.001$

$\delta = 0.2$

The output of the program should be as follows:

```
At time t = 0, there are 500 rabbits, and 1 foxes
At time t = 1, there are 598 rabbits, and 1 foxes
At time t = 2, there are 713 rabbits, and 2 foxes
At time t = 3, there are 849 rabbits, and 3 foxes
At time t = 4, there are 1007 rabbits, and 5 foxes
At time t = 5, there are 1186 rabbits, and 8 foxes
At time t = 6, there are 1375 rabbits, and 16 foxes
At time t = 7, there are 1538 rabbits, and 35 foxes
At time t = 8, there are 1573 rabbits, and 83 foxes
```

```
At time t = 9, there are 1237 rabbits, and 196 foxes

At time t = 10, there are 270 rabbits, and 400 foxes

At time t = 11, there are 0 rabbits, and 428 foxes

At time t = 12, there are 0 rabbits, and 342 foxes

At time t = 13, there are 0 rabbits, and 274 foxes

At time t = 14, there are 0 rabbits, and 219 foxes
```

(10 points)

A.6.2 How long can 100 rabbits survive?

Suppose at $t = 0$, there are 100 rabbits, i.e., $r_0 = 100$. How many foxes should be there at $t = 0$ (i.e., what should be f_0), such that the rabbit species survives (i.e., $r_{t\_max} > 0$) for the maximum possible number of years (t_{max}) before becoming extinct (i.e., $r_{t\_max+1} = 0$). Also, find the maximum possible number of years (i.e., t_{max}) the rabbit species will survive.

Modify the program in the previous question to compute the answers to the above questions, and print the following statement, with the blanks filled:

```
If there are ___ foxes at t = 0, the rabbit species will survive for ___
years, which is the maximum possible number of years the rabbits can survive.
```

Note: Use the same values of α, β, γ, and δ as in the previous question.

Hint:

1. Consider values of f_0 starting from 1, and upto a large number, say 1000.
2. For each value of f_0, find the number of years for which the rabbit species survives.
3. Find the value of f_0 and t for which the rabbit species surives the maximum number of years, i.e., $t = t_{max}$.

(20 points)

A.6.3 Saving rabbits from extinction

What must be the minimum number of rabbits, and the corresponding number of foxes at $t = 0$, such that the rabbit and fox species never become extinct.

Note: Use the same values of α, β, γ, and δ as in the previous question.

Hint:

1. Consider $r_0 = 1$, and then keep increasing r_0 by 1 if it's not possible for the rabbit species to survive with the value of r_0 under consideration.
2. For each r_0, consider number of foxes starting from $f_0 = 1$, and upto a large number, say $f_0 = 200$.
3. As soon as you find a combination of r_0 and f_0, such that there is no change in r_t and f_t for 2 consecutive years, you have found the values of r_0 and f_0, such that both the species maintain their numbers and never become extinct. At this point, print the result, and stop the program *(break out of all loops).*

Modify the program in the previous question to answer the above question, and print the following statement with the blanks filled:

```
For ___ foxes, and ___ rabbits at t = 0, the fox and rabbit species will
never be extinct.
```

(25 points)

B Datasets

Datasets used in the book can be found here