MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
SUMY STATE UNIVERSITY

U. Shvets, I. Knyaz’

Python and Data Science

Lecture notes

Sumy
Sumy State University
2021

MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
SUMY STATE UNIVERSITY

U. Shvets, I. Knyaz'

Python and Data Science

Lecture notes

APPROVED
at the session of Applied Mathematics and Complex Systems Modelling Department as lecture notes on discipline "Python and Data Science".
Minutes № 9 of 19.10.2021.

Sumy
Sumy State University
2021

Python and Data Science: lecture notes / compilers U. Shvets, I. Knyaz'. – Sumy : Sumy State University, 2021. – 101 p.

Department of Applied Mathematics and Complex Systems Modelling

CONTENTS

P.

INTRODUCTION TO PYTHON

Python is an easy to learn, powerful programming language. It has efficient high-level data structures. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms. The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python Web site, https://www.python.org/, and may be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation. The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C).

This lecture notes will introduce you to the basics of the python programming environment, including fundamental python programming techniques. This introduction to Python will kickstart your learning of **Python for data science**, as well as programming in general. The lecture notes will introduce data manipulation and cleaning techniques using the popular python data science libraries and introduce the abstraction of the Series and DataFrame as the central data structures for data analysis.

To write programs you may use IDLE – a simple integrated development environment (IDE) that comes with Python. You can find IDLE in the Python folder on your computer. When you first start IDLE, it starts up in the shell, which is an interactive window where you can type in Python code and see the output in the same window.

1.1 Simple Python program

Start IDLE and open up a new window (choose New Window under the File Menu). Type in the following program:

```
temp = eval(input('Enter a temperature in
                                    Celsius: '))
print('In Kelvin, that is', temp+273)
```

Then, under the Run menu, choose Run Module (or press F5). IDLE will ask you to save the file, and you should do so. Be sure to append .py to the filename as IDLE will not automatically append it. This will tell IDLE to use colors to make your program easier to read.

Once you've saved the program, it will run in the shell window. The program will ask you for a temperature. Type in 20 and press Enter. The program's output looks something like this:

```
Enter a temperature in Celsius: 20
In Fahrenheit, that is 293.0
```

Let's examine how the program does what it does. The first line asks the user to enter a temperature. The `input` function's job is to ask the user to type something in and to capture what the user types. The part in quotes is the prompt that the user sees. It is called a *string* and it will appear to the program's user exactly as it appears in the code itself. The `eval` function is something we use here, but it won't be clear exactly why until later. So for now, just remember that we use it when we're getting numerical input.

We need to give a name to the value that the user enters so that the program can remember it and use it in the second line. The name we use is `temp` and we use the equals sign to assign the user's value to `temp`.

The second line uses the `print` function to print out the conversion. The part in quotes is another string and will appear to your program's user exactly as it appears in quotes here. The second argument to the `print` function is the calculation. Python will do the calculation and print out the numerical result.

This program may seem too short and simple to be of much use, but there are many websites that have little utilities that do similar conversions, and their code is not much more complicated than the code here.

A second program here is a program that computes the average of three numbers that the user enters:

```
n1 = eval(input('Enter the first number:'))
n2 = eval(input('Enter the second number:'))
n3 = eval(input('Enter the third number:'))
print('The average of the numbers you entered
                          is', (n1+n2+n3)/3)
```

For this program we need to get three numbers from the user. There are ways to do that in one line, but for now we'll keep things simple. We get the numbers one at a time and give each number its own name. The only other thing to note is the parentheses in the average calculation. This is because of the order of operations. All multiplications and divisions are performed before any additions and subtractions, so we have to use parentheses to get Python to do the addition first[1)].

1.2 Getting input and printing results

The `input` function is a simple way for your program to get information from people using your program. Here is an example:

```
name = input('Enter your surname: ')
print('Hello ', surname)
```

The basic structure is

variable name = `input`(`'`*message to user*`'`)

The above works for getting text from the user. To get numbers

---

[1)] Spaces matter at the beginning of lines, but not elsewhere. For example, the code below will not work properly:

```
num1 = eval(input('Enter the first number:'))
       num2 = eval(input('Enter the second number:'))
              print('The average of the numbers you
                                 entered is', (num1+num2)/2)
```

from the user to use in calculations, we need to do something extra. Here is an example:

```
num = eval(input('Enter a number:    '))
print('Your number cubed: ', num*num*num)
```

The `eval` function converts the text entered by the user into a number. One nice feature of this is you can enter expressions, like `(3*2+3)/4`, and `eval` will compute them for you.
There are several useful functions for convertation. The `int` function converts something into an integer. The `float` function converts something into a floating point number. Here are some examples.

Statement	Result
`int('27')`	27
`float('3.14')`	3.14
`int(3.14)`	3

To convert a float to an integer, the `int` function drops everything after the decimal point.

So, the previous example we may rewrite with `float`:

```
num = float(input('Enter a number:    '))
print('Your number cubed: ', num*num*num)
```

Quite often we will want to convert a number to a string to take advantage of string methods to break the number apart. The built-in function `str` is used to convert things into strings. Here are some examples:

Statement	Result
`str(27)`	`'27'`
`str(3.14)`	`'3.14'`
`str([1,2,3])`	`'[1,2,3]`

The `print` function requires parenthesis around its arguments. Anything inside quotes will (with a few exceptions) be printed exactly as it appears. In the following, the first statement will output 3 + 4, while the second will output 7:

```
print('3+4')
print(3+4)
```

To print several things at once, separate them by commas. Python will automatically insert spaces between them. Below is an example and the output it produces.

```
print('The value of 2+2 is', 2+2)
print('A',   1)
```

As a result we will get

```
The value of 2+2 is 4
A 1
```

There are two optional arguments to the `print` function. Python will insert a space between each of the arguments of the print function. There is an optional argument called `sep`, short for separator, that you can use to change that space to something else. For example, using `sep=':'` would separate the arguments by a colon and `sep='##'` would separate the arguments by two pound signs. One particularly useful possibility is to have nothing inside the quotes, as in `sep=''`. This says to put no separation between the arguments. Here is an example where `sep` is useful for getting the output to look nice:

```
print('The value of 3+5 Is', 3+5,    '.')
print('The value of 3+5 Is', 3+5, sep=' ')
```

As a result we will get

```
The value of 3+5 is 8 .
The value of 3+5 is  8.
```

The print function will automatically advance to the next line. There is an optional argument called `end` that you can use to keep the print function from advancing to the next line. Here is an example:

```
print('On the first line', end='')
print('On the second line')
```

As a result we will get

```
On the first lineOn the second line
```

1.3 Variables

Looking back at our first program, we see the use of a variable called `temp`:

```
temp = eval(input('Enter a temperature in
                                    Celsius: '))
print{'In Kelvin, that is', temp+273)
```

One of the major purposes of a variable is to remember a value from one part of a program so that it can be used in another part of the program. In the case above, the variable `temp` stores the value that the user enters so that we can do a calculation with it in the next line.

Here is another example with variables:

```
x=3
y=4
z=x+y
z=z+l
x=y
y=5
```

After these four lines of code are executed, `x` is 4, `y` is 5 and `z` is 8. One way to understand something like this is to take it one line at a time. This is an especially useful technique for trying to understand

more complicated chunks of code. Here is a description of what happens in the code above:

1. `x` starts with the value 3 and `y` starts with the value 4.
2. In line 3, a variable `z` is created to equal `x+y`, which is 7.
3. Then the value of `z` is changed to equal one more than it currently equals, changing it from 7 to 8.
4. Next, `x` is changed to the current value of `y`, which is 4.
5. Finally, `y` is changed to 5. Note that this does not affect `x`.
6. So at the end, `x` is 4, `y` is 5, and `z` is 8.

There are just a couple of rules to follow when naming your variables.

- Variable names can contain letters, numbers, and the underscore.
- Variable names *cannot* contain spaces.
- Variable names *cannot* start with a number.
- Case matters – for instance, `temp` and `Temp` are different.

LECTURE 2

NUMBERS AND OPERATIONS WITH THEM

2.1 Integers and Decimal Numbers

Because of the way computer chips are designed, integers and decimal numbers are represented differently on computers. Decimal numbers are represented by what are called floating point numbers. The important thing to remember about them is you typically only get about 15 or so digits of precision. It would be nice if there were no limit to the precision, but calculations run a lot more quickly if you cut off the numbers at some point.

On the other hand, integers in Python have no restrictions. They can be arbitrarily large.

For decimal numbers, the last digit is sometimes slightly off due to the fact that computers work in binary (base 2) whereas our human number system is base 10. As an example, mathematically, we know that the decimal expansion of 7/3 is 2.333···, with the threes repeating forever. But when we type 7/3 into the Python shell, we get `2.3333333333333335`. This is called *roundoff error*. For most practical purposes this is not too big of a deal, but it actually can cause problems for some mathematical and scientific calculations.

2.2 Math Operators and Order of operations

Here is a list of the common operators in Python:

Operator	Description
+	Addition
−	Subtraction
*	Multiplication
/	Division

Operator	Description
`**`	Exponentiation
`//`	integer division e. g. 6//5 = 1
`%`	modulo (remainder) e. g. `18%7` = 4

Exponentiation gets done first, followed by multiplication and division (including `//` and `%`), and addition and subtraction come last.

This comes into play in calculating an average. Say you have three variables `x`, `y`, and `z`, and you want to calculate the average of their values and to add value of *f*. To expression `x+y+z/3+f` would not work. Because division comes before addition, you would actually be calculating $x + y + \frac{z}{3} + f$ instead of $\frac{x+y+z}{3} + f$. This is easily fixed by using parentheses: `(x+y+z)/3+f`.

In general, if you're not sure about something, adding parentheses might help and usually doesn't do any harm.

2.3 Random numbers

Python comes with a module, called `random`, that allows us to use random numbers in our programs.

Before we get to random numbers, we should first explain what a *module* is. The core part of the Python language consists of things like for loops, if statements, math operators, and some functions, like `print` and `input`. Everything else is contained in modules, and if we want to use something from a module we have to first *import* it — that is, tell Python that we want to use it.

At this point, there is only one function, called `randint`, that we will need from the `random` module. To load this function, we use the following statement:

```
from random import randint
```

Using `randint` is simple: `randint(a,b)` will return `a` random integer between `a` and `b` including both `a` and `b`. (Note that `randint` includes the right endpoint `b` unlike the `range` function). Here is a short example:

```
from random import randint
x = randint(1,20)
print('A random number between 1 and 20: ',x)
```

As a result we will get:

```
A random number between 1 and  20: 7
```

The random number will be different every time we run the program.

2.4 Math module

Python has a module called `math` that contains familiar math functions. There are also the inverse trig functions, hyperbolic functions, and the constants `pi` and `e`. Here is the list of all the functions and attributes defined in math module with a brief explanation of what they do.

List of Functions in Python Math Module

Function	Description
`ceil(x)`	Returns the smallest integer greater than or equal to `x`
`copysign(x,y)`	Returns `x` with the sign of `y`
`fabs(x)`	Returns the absolute value of `x`
`factorial(x)`	Returns the factorial of `x`
`floor(x)`	Returns the largest integer less than or equal to `x`
`fmod(x,y)`	Returns the remainder when `x` is divided by `y`
`frexp(x)`	Returns the mantissa and exponent of `x` as

Function	Description
	the pair (m, e)
`fsum(iterable)`	Returns an accurate floating point sum of values in the iterable
`isfinite(x)`	Returns True if `x` is neither an infinity nor a NaN (Not a Number)
`isinf(x)`	Returns True if `x` is a positive or negative infinity
`isnan(x)`	Returns True if `x` is a NaN
`ldexp(x,i)`	Returns `x*(2**i)`
`modf(x)`	Returns the fractional and integer parts of `x`
`trunc(x)`	Returns the truncated integer value of `x`
`exp(x)`	Returns `e**x`
`expm1(x)`	Returns `e**x-1`
`log(x[,b])`	Returns the logarithm of `x` to the base `b` (defaults to `e`)
`log1p(x)`	Returns the natural logarithm of `1+x`
`log2(x)`	Returns the base-2 logarithm of `x`
`log10(x)`	Returns the base-10 logarithm of `x`
`pow(x,y)`	Returns `x` raised to the power `y`
`sqrt(x)`	Returns the square root of `x`
`acos(x)`	Returns the arc cosine of `x`
`asin(x)`	Returns the arc sine of `x`
`atan(x)`	Returns the arc tangent of `x`
`atan2(y,x)`	Returns atan `(y/x)`
`cos(x)`	Returns the cosine of `x`
`hypot(x,y)`	Returns the Euclidean norm, sqrt `(x*x+y*y)`
`sin(x)`	Returns the sine of `x`
`tan(x)`	Returns the tangent of `x`
`degrees(x)`	Converts angle `x` from radians to degrees
`radians(x)`	Converts angle `x` from degrees to radians
`acosh(x)`	Returns the inverse hyperbolic cosine of `x`

Function	Description
`asinh(x)`	Returns the inverse hyperbolic sine of `x`
`atanh(x)`	Returns the inverse hyperbolic tangent of `x`
`cosh(x)`	Returns the hyperbolic cosine of `x`
`sinh(x)`	Returns the hyperbolic cosine of `x`
`tanh(x)`	Returns the hyperbolic tangent of `x`
`erf(x)`	Returns the error function at `x`
`erfc(x)`	Returns the complementary error function at `x`
`gamma(x)`	Returns the Gamma function at `x`
`lgamma(x)`	Returns the natural logarithm of the absolute value of the Gamma function at `x`
`pi`	Mathematical constant, the ratio of circumference of a circle to it's diameter (`3.14159...`)
`e`	Mathematical constant e (`2.71828...`)

Here is a short example:

```
import math
number = -21.19
print('The given number is :', number)
print('Floor value is :', math.floor(number))
print('Ceiling value is :', math.ceil(number))
print('Absolute value is :', math.fabs(number))
```

There are two built in math functions, `abs` (absolute value) and `round` that are available without importing the math module. Here are some examples:

```
print(abs(-2.3))
print(round(5.436, 2))
print(round(116.2, -1))
```

As a result we will get

```
2.3
5.43
120.0
```

The `round` function takes two arguments: the first is the number to be rounded and the second is the number of decimal places to round to. The second argument can be negative.

LECTURE 3

CONDITIONAL FLOW CONTROL

3.1 Simple Conditions

Let us consider simple arithmetic comparisons that directly translate from math into Python

```
2 < 5
3 > 7
x = 11
x > 10
2 * x < x
type(True)
```

You see that conditions are either `True` or `False`. These are the only possible Boolean values. In Python the name Boolean is shortened to the type `bool`. It is the type of the results of true-false conditions or tests.

3.2 Simple *if* Statements

Run this example program. Try it at least twice, with inputs: 30 and then 55. As you see, you get an extra result, depending on the input. The main code is:

```
weight = float(input("How many pounds does
                      your suitcase weigh? "))
if weight > 50:
    print("There is a $25 charge for luggage
                                  that heavy.")
print("Thank you for your business.")
```

The middle two line are an `if` statement. If it is true that the weight is greater than 50, then print the statement about an extra charge. If it is not true that the weight is greater than 50, then don't do the indented part: skip printing the extra luggage charge. In any event, when you have finished with the `if` statement (whether it actually does anything or not), go on to the next statement that is *not* indented under the `if`. In this case that is the statement printing "Thank you".

The general Python syntax for a simple `if` statement is

```
if condition :
     indented Statement Block
```

If the condition is true, then do the indented statements. If the condition is not true, then skip the indented statements.

In the examples above the choice is between doing something (if the condition is `True`) or nothing (if the condition is `False`). Often there is a choice of two possibilities, only one of which will be done, depending on the truth of a condition.

3.3 *if-else* Statements

Try the program below at least twice, with inputs 50 and then 80. As you can see, you get different results, depending on the input.

```
temperature = float(input('What is the
                                temperature? '))
if temperature > 70:
     print('Wear shorts')
else:
     print('Wear long pants')
print('Get some exercise outside')
```

The middle four lines are an `if-else` statement. Again it is close to English, though you might say "otherwise" instead of "else" (but else is shorter!). There are two indented blocks: One, like in the simple `if` statement, comes right after the `if` heading and is executed when the

condition in the `if` heading is `true`. In the `if-else` form this is followed by an else: line, followed by another indented block that is only executed when the original condition is `false`. In an `if-else` statement exactly one of two possible indented blocks is executed.

The general Python `if-else` syntax is

```
if condition :
     indented Statement Block For True
                                      Condition
else:
     indented Statement Block For False
                                      Condition
```

These statement blocks can have any number of statements, and can include about any kind of statement.

All the usual arithmetic comparisons may be made, but many do not use standard mathematical symbolism, mostly for lack of proper keys on a standard keyboard.

Meaning	Math Symbol	Python Symbols
Less than	<	<
Greater than	>	>
Less than or equal	≤	<=
Greater than or equal	≥	>=
Equals	=	==
Not equal	≠	!=

There should not be space between the two-symbol Python substitutes.

Tests for equality do not make an assignment, and they do not require a variable on the left. Any expressions can be tested for equality or inequality (!=). They do not need to be numbers! Predict the results for these commands:

```
x = 5
x == 5
x != 6
x = 6
6 == x
6 != x
'hi' == 'h' + 'i'
'HI' != 'hi'
[1, 2] != [2, 1]
'a' > 5
```

There are three additional operators used to construct more complicated conditions: `and`, `or`, and `not`. Here are an example:

```
if(score<100 or time>2) and turns_remain==0:
     print('Game over.')
```

In terms of order of operations, `and` is done before `or`, so if you have a complicated condition that contains both, you may need parentheses around the `or` condition. Think of and as being like multiplication `and` or as being like addition.

On the other hand, there is a nice shortcut that does work in Python (though not in many other programming languages):

```
if 80<=score<90:
     …
```

Here are some handy shortcuts:

Statement	Shortcut
`if a==0 and b==0 and c==0:`	`if a==b==c==0:`
`if 1<a and a<b and b<5:`	`if 1<a<b<5:`

3.4 Multiple Tests and *if-elif* Statements

Often you want to distinguish between more than two distinct cases, but conditions only have two possible results, `True` or `False`, so the

only direct choice is between two options. As anyone who has played "20 Questions" knows, you can distinguish more cases by further questions. If there are more than two choices, a single test may only reduce the possibilities, but further tests can reduce the possibilities further and further. Since most any kind of statement can be placed in an indented statement block, one choice is a further if statement. For instance consider a function to convert a numerical grade to a letter grade, 'A', 'B', 'C', 'D' or 'F', where the cutoffs for 'A', 'B', 'C', and 'D' are 90, 80, 70, and 60 respectively. One way to write the function would be test for one grade at a time, and resolve all the remaining possibilities inside the next else clause:

```
if score >= 90:
     letter = 'A'
else:   # grade must be B, C, D or F
     if score >= 80:
          letter = 'B'
     else:  # grade must be C, D or F
          if score >= 70:
               letter = 'C'
          else:    # grade must D or F
               if score >= 60:
                    letter = 'D'
               else:
                    letter = 'F
print(letter)
```

This repeatedly increasing indentation with an `if` statement as the `else` block can be annoying and distracting. A preferred alternative in this situation, that avoids all this indentation, is to combine each `else` and `if` block into an `elif` block:

```
if score >= 90:
     letter = 'A'
elif score >= 80:
     letter = 'B'
elif score >= 70:
```

```
     letter = 'C'
elif score >= 60:
     letter = 'D'
else:
     letter = 'F'
```

The most elaborate syntax for an `if-elif-else` statement is indicated in general below:

```
if condition1 :
     indented Statement Block For True
                                    Condition 1
elif condition2 :
     indented Statement Block For First True
                                    Condition 2
elif condition3 :
     indented Statement Block For First True
                                    Condition 3
elif condition4 :
     indented Statement Block For First True
                                    Condition 4
else:
     indented Statement Block For Each
                                Condition False
```

The `if`, each `elif`, and the final `else` lines are all aligned. There can be any number of `elif` lines, each followed by an indented block. (Three happen to be illustrated above.) With this construction *exactly one* of the indented blocks is executed. It is the one corresponding to the *first* `True` condition, or, if all conditions are `False`, it is the block after the final `else` line.

LECTURE 4

LOOPS

Loops are among the most basic and powerful of programming concepts. A loop in a computer program is an instruction that repeats until a specified condition is reached. In a loop structure, the loop asks a question. If the answer requires action, it is executed. The same question is asked again and again until no further action is required. Each time the question is asked is called an iteration. A computer programmer who needs to use the same lines of code many times in a program can use a loop to save time.

4.1 *for* loop

The following program will print `Hello` and `How are you?` twenty times:

```
for i in range(20):
     print ('Hello')
     print ('How are you?')
```

The structure of a `for` loop is as follows:

```
for variable name in range(number of times to
                                        repeat):
    statements to be repeated
```

The syntax is important here. The word `for` must be in lowercase, the first line must end with a colon, and the statements to be repeated ***must* be indented**. Indentation is used to tell Python which statements will be repeated.

Since the second and third lines are indented, Python knows that these are the statements to be repeated. If a line is not indented, so

it is not part of the loop and only gets executed once, after the loop has completed.

Looking at the above example, we see where the term *for loop* comes from: we can picture the execution of the code as starting at the `for` statement, proceeding to the second and third lines, then looping back up to the `for` statement.

There is one part of a for loop that is a little tricky, and that is the loop variable. In the example below, the loop variable is the variable i. The output of this program will be the numbers 0,1,..., 99, each printed on its own line.

```
for i in range(100):
    print(i)
```

When the loop first starts, Python sets the variable `i` to 0. Each time we loop back up, Python increases the value of `i` by 1. The program loops 100 times, each time increasing the value of `i` by until we have looped 100 times. At this point the value of i is 99.

You may be wondering why `i` starts with 0 instead of 1. Well, there doesn't seem to be any really good reason why other than that starting at 0 was useful in the early days of computing and it has stuck with us. In fact most things in computer programming start at 0 instead of 1. This does take some getting used to.

It's a convention in programming to use the letters i, j, and k for loop variables, unless there's a good reason to give the variable a more descriptive name.

4.2 The range function

The value we put in the `range` function determines how many times we will loop. The way `range` works is it produces a list of numbers from zero to the value minus one. For instance, `range(5)` produces five values: 0, 1, 2, 3, and 4.

If we want the list of values to start at a value other than 0, we can do that by specifying the starting value. The statement `range(1,5)` will produce the list 1, 2, 3, 4. This brings up one quirk

of the `range` function - it stops one short of where we think it should. If we wanted the list to contain the numbers 1 through 5 (including 5), then we would have to do `range(1,6)`.

Another thing we can do is to get the list of values to go up by more than one at a time. To do this, we can specify an optional step as the third argument. The statement `range(1,10,2)` will step through the list by twos, producing 1, 3, 5, 7, 9.

To get the list of values to go backwards, we can use a step of -1. For instance, `range (5,1,-1)` will produce the values 5, 4, 3, 2, in that order. Note that the `range` function stops one short of the ending value 1. Here are a few more examples:

Statement	Values generated
`range(8)`	`0,1,2,3,4,5,6,7`
`range(1,8)`	`1,2,3,4,5,6,7`
`range(2,7)`	`2,3,4,5,6`
`range(2,13,2)`	`2,4,6,8,10,12`
`range(9,2,-1)`	`9,8,7,6,5,4,3`

Here is an example program that counts down from 5 and then prints a message.

```
for i in range(4,0,-1):
     print (i, end=' ')
print('Stop')
```

```
         4 3 2 1 Stop
```

The `end=' '` just keeps everything on the same line.

The program below prints a triangle of symbols ‘o’ that is 4 rows tall

```
for i in range(4):
     print('o'*(i+1))
```

The triangle produced by this code is shown below

```
o
oo
ooo
oooo
```

The code `'o'*(i+1)` is something we'll cover in next lecture; it just repeats character `'o'` six times.

Sometimes, though, we need to repeat something, but we don't know ahead of time exactly how many times it has to be repeated. For instance, a game of Tic-tac-toe keeps going until someone wins or there are no more moves to be made, so the number of turns will vary from game to game. This is a situation that would call for a while loop.

4.3 *while* loop

Let's go back to the first program we wrote back in Lecture 1, the temperature converter. One annoying thing about it is that the user has to restart the program for every new temperature. A while loop will allow the user to repeatedly enter temperatures. A simple way for the user to indicate that they are done is to have them enter a nonsense temperature like 66. This is done below:

```
temp = 0
while temp!=66:
     temp = eval(input('Enter a temperature
                              (66 to quit):'))
     print('In Kelvin, that is', temp+273)
```

Look at the `while` statement first. It says that we will keep looping, that is, keep getting and converting temperatures, as long as the temperature entered is not 66. As soon as 66 is entered, the while loop stops. Tracing through, the program first compares `temp` to 66. If `temp` is not 66, then the program asks for a temperature and converts it. The program then loops back up and again compares `temp` to 66. If `temp` is not 66, the program will ask for another temperature,

convert it, and then loop back up again and do another comparison. It continues this process until the user enters 66.

We need the line `temp=0` at the start, as without it, we would get a name error. The program would get to the `while` statement, try to see if `temp` is not equal to 66 and run into a problem because temp doesn't yet exist. To take care of this, we just declare `temp` equal to 0. There is nothing special about the value 0 here. We could set it to anything except 66. (Setting it to 66 would cause the condition on the while loop to be false right from the start and the loop would never run.)

Note that is natural to think of the while loop as continuing looping until the user enters 66. However, when we construct the condition, instead of thinking about when to stop looping, we instead need to think in terms of what has to be true in order to keep going.

We can use a `while` loop to mimic a `for` loop, as shown below. Both loops have the exact same effect:

```
for i in range(10):
    print(i)
```

```
i=0
while i<10:
    print(i)
    i=i+1
```

Remember that the `for` loop starts with the loop variable `i` equal to 0 and ends with it equal to 9. To use a `while` loop to mimic the `for` loop, we have to manually create our own loop variable `i`. We start by setting it to 0. In the `while` loop we have the same `print` statement as in the `for` loop, but we have another statement, `i=i+1`, to manually increase the loop variable, something that the `for` loop does automatically.

When working with while loops, sooner or later you will accidentally send Python into a never-ending loop. Here is an example:

```
while True:
    # statements to be repeated go here
```

In this program, the condition is always true (you may replace `True` by any positive number, e. g. `while 1`). Python will continuously repeat intended statements below `while`. To stop a program caught in a never-ending loop, use `Restart Shell` under the `Shell` menu. You can use this to stop a Python program before it is finished executing.

Another example of an infinite loop is here:

```
i=0
n=1
while i<1 and n>0:  # Conditions are always true
     print(0)
```

The `break` statement can be used to break out of a `for` or `while` loop before the loop is finished.

Here is a program that allows the user to enter up to 10 numbers. The user can stop early by entering a negative number.

```
for i in range(10):
     n = eval(input('Enter a number:'))
     if n<0:
           break
```

When a for loop is present inside another for loop then it is called a nested for loop. Let's take an example of nested for loop.

```
for num1 in range(3):
     for num2 in range(10, 14):
          print(num1, ",", num2)
```

As a result we will get:

```
0,10
0,11
0,12
0,13
```

```
1,10
1,11
1,12
1,13
2,10
2,11
2,12
2,13
```

Unlike other languages we can use `else` for loops. When the loop condition of "for" or "while" statement fails then code part in "else" is executed. If a `break` statement is executed inside the `for` loop then the "else" part is skipped. Note that the "else" part is executed even if there is a `continue` statement. Let's consider another that allows us to print out 0, 1, 2, 3, 4 and then print "count value reached 5".

```
count=0
while(count<5):
    print(count)
    count +=1
else:
    print("count value reached %d" %(count))

# Prints out 1,2,3,4
for i in range(1, 10):
    if(i%5==0):
        break
    print(i)
else:
    print("this is not printed because for
loop is terminated because of break but not
due to fail in condition")
```

LECTURE 5

STRING TREATMENT

Strings are a data type in Python for dealing with text. Python has a number of powerful features for manipulating strings.

A string is created by enclosing text in quotes. You can use either single quotes, `'`, or double quotes, `"`. A triple-quote can be used for multi-line strings. Here are some examples:

```
s = 'Hi'
t = "Hello"
I = """This Is a long string that is
spread across two lines."""
```

In previous lectures when getting numerical input we used an `eval` statement with the `input` statement, but when getting text, we do not use `eval`.

The empty string `''` is the string equivalent of the number 0. It is a string with nothing in it. We have seen it before, in the print statement's optional argument, `sep=''`.

To get the length of a string (how many characters it has), use the built-in function `len`. For example, `len('How are you')` is 11 (spaces are symbols too).

The operators + and `*` can be used on strings. The + operator combines two strings. This operation is called *concatenation.* The `*` repeats a string a certain number of times. Here are some examples.

Expression	Result
`'AB'+'cd'`	`'ABcd'`
`'A'+'7'+'B'`	`'A7B'`
`'Hi'*4`	`'HiHiHiHi'`

The `in` operator is used to tell if a string contains something. For example:

```
string='Hello. How are you?'
if 'H' in string:
     print('Your string contains H')
```

We will often want to pick out individual characters from a string. Python uses square brackets to do this. The table below gives some examples of indexing the string `s='Python'`.

Statement	Result	Description
`s[0]`	`P`	first character of `s`
`s[1]`	`y`	second character of `s`
`s[-1]`	`n`	last character of `s`
`s[-2]`	`o`	second-to-last character of `s`

Note, that the first character of `s` is `s[0]`, not `s[1]`. Remember that in programming, counting usually starts at 0, not 1. Negative indices count backwards from the end of the string.

A common error Suppose `s='Python'` and we try to do `s[12]`. There are only six characters in the string and Python will raise the following error message:

```
IndexError: string index out of range
```

A *slice* is used to pick out part of a string. It behaves like a combination of indexing and the `range` function. Below we have some examples with the string `s='abcdefghij'`.

Code	Result	Description
`s[2:5]`	`cde`	characters at indices 2, 3, 4
`s[ :5]`	`abcde`	first five characters
`s[5: ]`	`fghij`	characters from index 5 to the end

Code	Result	Description
`s[-2: ]`	`ij`	last two characters
`s[ : ]`	`abcdefghij`	entire string
`s[1:7:2]`	`bdf`	characters from index 1 to 6, by twos
`s[ : :-1]`	`jihgfedcba`	a negative step reverses the string

Slices have the same quirk as the `range` function in that they does not include the ending location. For instance, in the example above, `s[2:5]` gives the characters in indices 2, 3, and 4 but not the character in index 5.

We can leave either the starting or ending locations blank. If we leave the starting location blank, it defaults to the start of the string. So `s[:5]` gives the first five characters of s. If we leave the ending location blank, it defaults to the end of the string. So `s[5:]` will give all the characters from index 5 to the end. If we use negative indices, we can get the ending characters of the string. For instance, `s[-2:]` gives the last two characters.

There is an optional third argument, just like in the `range` statement, that can specify the step. For example, `s[1:7:2]` steps through the string by twos, selecting the characters at indices 1,3, and 5 (but not 7, because of the aforementioned quirk). The most useful step is -1, which steps backwards through the string, reversing the order of the characters.

Suppose we have a string called `s` and we want to change the character at index 4 of `s` to `'X'`. It is tempting to try `s[4]='X'`, but that unfortunately will not work. Python strings are *immutable,* which means we can't modify any part of them. If we want to change a character of `s`, we have to instead build a new string from `s` and reassign it to `s`. Here is code that will change the character at index 4 to `'X'`:

```
s = s[:4] + 'X' + s[5:]
```

The idea of this is we take all the characters up to index 4, then

X, and then all of the characters after index 4.

Very often we will want to scan through a string one character at a time. A for loop like the one below can be used to do that. It loops through a string called s, printing the string, character by character, each on a separate line:

```
for i in range(len(s)):
    print(s[i])
```

In the range statement we have len(s) that returns how long s is. So, if s were 5 characters long, this would be like having range(5) and the loop variable i would run from 0 to 4. This means that s[i] will run through the characters of s. This way of looping is useful if we need to keep track of our location in the string during the loop.

If we don't need to keep track of our location, then there is a simpler type of loop we can use:

```
for c in s:
    print(c)
```

This loop will step through s, character by character, with c holding the current character.

Strings come with a ton of *methods*, functions that return information about the string or return a new string that is a modified version of the original. Here are some of the most useful ones:

Method	Description
lower()	returns a string with every letter of the original in lowercase
upper()	returns a string with every letter of the original in uppercase
replace(x,y)	returns a string with every occurrence of x replaced by y

Method	Description
`count(x)`	counts the number of occurrences of `x` in the string
`index(x)`	returns the location of the first occurrence of `x`
`isalpha()`	returns `True` if every character of the string is a letter

One **very important note** about `lower`, `upper`, and `replace` is that they do not change the original string. If you want to change a string, `s`, to all lowercase, it is not enough to just use `s.lower()`. You need to do the following:

```
s = s.lower()
```

Here are some examples of string methods in action:

Statement	Description
`print(s.count(''))`	prints the number of
`s=s.upper()`	changes the string to all
`s=s.replace('Hi','Hello')`	replaces each `'Hi'` in `s`
`print(s.index('a'))`	prints location of the first

The `isalpha` method is used to tell if a character is a letter or not. It returns `True` if the character is a letter and `False` otherwise. When used with an entire string, it will only return `True` if every character of the string is a letter. The values `True` and `False` are called Booleans. For now, though, just remember that you can use `isalpha` in if conditions.

If you try to find the index of something that is not in a string, Python will raise an error. For instance, if `s='abc'` and you try `s.index('z')`, you will get an error.

Let's consider an example. Replace all occurrences of "is" with "WAS"

```
string = "This is nice. This is good."
newString = string.replace("is","WAS")
print(newString)
```

As a result we will get
```
ThWAS WAS nice. ThWAS WAS good.
```

Let's write a program that asks the user for a string and prints out the location of each `'e'` in the string and calculates a number of locations.

We use a loop to scan through the string one character at a time. The loop variable `i` keeps track of our location in the string, and `s[i]` gives the character at that location. Thus, the third line checks each character to see if it is an `'a'`, and if so, it will print out `i`, the location of that `'a'`.

```
s = input('Enter a sentence: ')
num=0
for i in range(len(s)):
     if s[i]=='e':
          print(i)
          num=num+1
```

Here is another example. Let's consider a very old method of sending secret messages based on the substitution cipher. Basically, each letter of the alphabet gets replaced by another letter of the alphabet, say every a gets replaced with an `x`, and every `b` gets replaced by a `z`, etc.

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'
key = 'xznlwebgjhqdyvtkfuompciasr'
message = input('Enter your text: ').lower()
for c in message:
     if c.isalpha():
          print(key[alphabet.index(c)],end='')
     else:
          print(c, end='')
```

The string `key` is a random reordering of the alphabet.

The only tricky part of the program is the `for` loop. What it does is go through the message one character at a time, and, for every letter it finds, it replaces it with the corresponding letter from the key. This is accomplished by using the `index` method to find the position in the alphabet of the current letter and replacing that letter with the letter from the key at that position. All non-letter characters are copied as is. The program uses the `isalpha` method to tell whether the current character is a letter or not.

The code to decipher a message is nearly the same. Just change `key[alphabet.index(c)]` to `alphabet[key.index(c)]`.

LECTURE 6

PYTHON LISTS

Lists are used to store multiple items in a single variable. Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are `Tuple`, `Set`, and `Dictionary`, all with different qualities and usage.
Here is a simple list:

```
L=[4,1,4,7,8]
```

Use square brackets to indicate the start and end of the list, and separate the items by commas.

The empty list is `[]`. It is the list equivalent of `0` or `''`. We can use `eval(input())` to allow the user to enter a list and the `print` function to print the entire contents of a list.. Here is an example:

```
L = eval(input ('Enter a list: '))
print(L)
```

Lists can contain all kinds of things, even other lists. For example, the following is a valid list:

```
['physics', 'chemistry', 1997, 2000]
```

Making copies of lists is a little tricky due to the way Python handles lists. Say we have a list `L` and we want to make a copy of the list and call it `M`. For now, do the following in place of `M=L`:

```
M = L[:]
```

There are a number of things which work the same way for lists as for

strings. E. g.:

- `len` – the number of items in L is given by `len(L)`.
- `in` – the `in` operator tells you if a list contains something.

Here are some examples:

```
if 'physics' in L:
     print('Your list contains
                                  'physics'.')
```

- Indexing and slicing. These work exactly as with strings. For example, `L[0]` is the first item of the list L and `L[:3]` gives the first three items.
- `index` and `count`. These methods work the same as they do for strings.
- `+` and `*`. The `+` operator adds one list to the end of another. The `*` operator repeats a list. Here are some examples:

Expression	Result
`[1,2]+[3,4,5]`	`[1,2,3,4,5]`
`[2,3]*3`	`[2,3,2,3,2,3]`
`[0]*5`	`[0,0,0,0,0]`

The last example is particularly useful for quickly creating a list of zeroes.

- Looping. The same two types of loops that work for strings also work for lists. Both of the following examples print out the items of a list, one-by-one, on separate lines.

```
for i in range(len(L)):
     print(L[i])
```

```
for item in L:
       print(item)
```

The left loop is useful for problems where you need to use the loop variable `i` to keep track of where you are in the loop. If that is not needed, then use the right loop, as it is a little simpler.

There are several built-in functions that operate on lists. Here are some useful ones:

Function	Description
`len`	returns the number of items in the list
`sum`	returns the sum of the items in the list
`min`	returns the minimum of the items in the list
`max`	returns the maximum of the items in the list

For example, the following computes the average of the values in a list `L`:

```
average = sum(L)/len(L)
```

Let's consider a program which allows us to count the number of strings where the string length is 2 or more and the first and last character are same from a given list of strings.

```
list=(['mk','abc', 'xyz', 'aba', '3344'])
num = 0
for word in list:
     if len(word) > 1 and word[0] == word[-1]:
          num += 1
print(num)
```

Output: `2`

Here are some list methods:

Method	Description
`append(x)`	adds `x` to the end of the list
`sort()`	sorts the list
`count(x)`	returns the number of times `x` occurs in the list
`index(x)`	returns the location of the first occurrence of `x`
`reverse()`	reverses the list

Method	Description
`remove(x)`	removes first occurrence of `x` from the list
`pop(p)`	removes the item at index `p` and returns its value
`insert(p,x)`	inserts `x` at index `p` of the list

`split()`	Returns a list of the words of a string. The method assumes that words are separated by whitespace, which can be either spaces, tabs or newline characters. Here is an example:
	`s = 'My name is Ivan'`
	`print(s.split())`
	Result:
	`['My','name','is','Ivan']`

`join(L)`	takes a list `L` of strings and joins them together into a single string

There is a big difference between list methods and string methods: String methods do not change the original string, but list methods do change the original list. To sort a list `L`, just use `L.sort()` and not `L=L.sort()`. In fact, the latter will not work at all.

Let's consider a program that takes a sequence of numbers and returns a list containing the square root of each number:

```
import math
numbers = [1, 4, 9, 16, 25, 36, 49, 64, 81]
result = []
for number in numbers:
     result.append(math.sqrt(number))
print(result)
```

Output:

```
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

Changing a specific item in a list is easier than with strings. To change the value in location 1 of `L` to 10, we simply say `L[1]=10`. If we want to insert the value 10 into location without overwriting what is currently there, we can use the `insert` method. To delete an entry from a list, we can use the `del` operator. Some examples are shown below. Assume `L=[6,7,8]` for each operation.

Operation	New L	Description
`L[1] = 9`	`[6,9,8]`	replace item at index 1 with 9
`L.insert(1,9)`	`[6,9,7,8]`	insert a 9 at index 1 without replacing
`del L[1]`	`[6,8]`	delete second item
`del L[:2]`	`[8]`	delete first two items

There is a module called string that contains, among other things, a string variable called `punctuation` that contains common punctuation. We can remove the punctuation from a string `s` with the following code:

```
from string import punctuation
for c in punctuation:
     s = s.replace(c, '')
```

The `join` method is in some sense the opposite of `split`. It is a string method that takes a list of strings and joins them together into a single string. Here are some examples, using the list
`L = ['A','B','C']`

Operation	Result
`' '.join(L)`	`A B C`
`''.join(L)`	`ABC`
`', '.join(L)`	`A, B, C`
`'***'.join(L)`	`A***B***C`

A powerful way to create lists **is using** `for` **loops**. Here is a simple example:

```
L = [i for i in range(5)]
```

This creates the list `[0,1,2,3,4]`. Notice that the syntax of a list comprehension is somewhat reminiscent of set notation in mathematics. Another example:

```
str='HELLO'
L=[c*2 for c in str]
L
```

Output:

```
['HH','EE',LL','LL','OO']
```

Another example:

```
sentence = 'the rocket came back from mars'
vowels = [i for i in sentence if i in 'aeiou']
vowels
```

Output:

```
['e', 'o', 'e', 'a', 'e', 'a', 'o', 'a']
```

You can use more than one `for` in a list comprehension:

```
L = [[i,j] for i in range(2) for j in
                                    range(2)]
```

We will get

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

This is the equivalent of the following code:

```
L=[]
for i in range(2):
     for j in range(2):
          L.append([i,j])
```

There are a number of common things that can be represented by two-dimensional list. In Python, one way to create a two-dimensional list is to create a list whose items are themselves lists. Here is an example:

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

We use two indices to access individual items, e. g., L[1][2] to get the entry in row 1 and column 2. To print a two-dimensional list, you can use nested `for` loops:

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

Nested for loops, like the ones used in printing a two-dimensional list, can also be used to process the items in a two-dimensional list. Here is an example that counts how many entries in a 10 x 5 list are even.

```
count = 0
for i in range(10):
     for j in range(5):
          if L[i][j]%2==0:
               count = count + 1
```

To create a larger list, you can use a list comprehension like below:

```
L = [[0]*300 for i in range(10)]
```

This creates a list of zeroes with 10 rows and 300 columns.

LECTURE 7

SUBPROGRAM: FUNCTIONS

Functions are useful for breaking up a large program to make it easier to read and maintain. They are also useful if find yourself writing the same code at several different points in your program. You can put that code in a function and call the function whenever you want to execute that code. You can also use functions to create your own utilities, math functions, etc.

Functions are defined with the `def` statement. The statement ends with a colon, and the code that is part of the function is indented below the `def` statement. Here we create a simple function that just prints something.

```
def my_print():
     print('Student')
my_print()
print(' Ivan')
my_print()
print(' Alexey')
my_print()
```

Output:

```
Student
 Ivan
Student
 Alexey
Student
```

The first two lines define the function. In the last five lines we call the function three times.

One use for functions is if you are using the same code over and over again in various parts of your program, you can make your program shorter and easier to understand by putting the code in a

function. For instance, suppose for some reason you need to print a box of stars like the one below at several points in your program.

Put the code into a function, and then whenever you need a box, just call the function rather than typing several lines of redundant code. Here is the function.

```
def recta():
     print('o'*15)
     print('o', ' '*11, 'o')
     print('o', ' '*11, 'o')
     print('o', ' '*11, 'o')
     print('o', ' '*11, 'o')
     print ('o'*15)
```

Output:

```
ooooooooooooooo
o             o
o             o
o             o
o             o
ooooooooooooooo
```

One benefit of this is that if you decide to change the size of the box, you just have to modify the code in the function, whereas if you had copied and pasted the box-drawing code everywhere you needed it, you would have to change all of them.

We can pass values to functions. Here is an example:

```
def recta(n,m):
     print('o'*n)
     print('o', ' '*m, 'o')
     print('o', ' '*m, 'o')
     print('o', ' '*m, 'o')
     print('o', ' '*m, 'o')
     print ('o'*n)
```

When we call the `recta(8,4)` function with the values 8 and 4, that value gets stored in the variables `n` and `m`. We can then refer to that variables `n` and `m` in our function's code.

Output:

```
oooooooo
o      o
o      o
o      o
o      o
oooooooo
```

We can write functions that perform calculations and return a result. Here is a simple function that solves quadratic equations:

```
from math import sqrt
def roots(a,b,c):
     disc = b**2 - 4*a*c
     if disc >= 0:
     return ("x= ",(-b + sqrt(disc))/(2*a),
               "x= ",(-b - sqrt(disc))/(2*a))
     if disc < 0:
          return ("x= ",-b/(2*a),"+",
               sqrt(disc*(-1))/(2*a),"i"
               "x= ",-b/(2*a),"-",
                    sqrt(disc*(-1))/(2*a),"i")
# --- The end of function's code --
a=float(input("a= "))
b=float(input("b= "))
c= float(input("c= "))
print(roots(a,b,c))
```

The `return` statement is used to send the result of a function's calculations back to the caller.

Notice that the function itself does not do any printing. The printing is done outside of the function. That way, we can do math

with the result, like below.

```
print(roots(a+2,b-1,c))
```

If we had just printed the result in the function instead of returning it, the result would have been printed to the screen and forgotten about, and we would never be able to do anything with it.

A `return` statement by itself can be used to end a function early.

```
def Chk(string, wrong_words):
     if string in wrong_words:
          return
     print('Warning!!!')

Chk('root',['table','luck','pen'])
```

Output:

```
Warning!!!
```

The same effect can be achieved with an if/else statement, but in some cases, using `return` can make your code simpler and more readable.

7.1 Default and Keyword arguments

You can specify a default value for an argument. This makes it optional, and if the caller decides not to use it, then it takes the default value. Here is an example:

```
def mult_print(string, n=2) :
     print(string * n)

mult_print('Student', 4)
mult_print('Student')
```

Output:

```
StudentStudentStudentStudent
StudentStudent
```

Default arguments need to come at the end of the function definition, after all of the non-default arguments.

A related concept to default arguments is *keyword arguments*. Say we have the following function definition:

```
def fancy_print(text, color, background,
                                style, justify):
```

Every time you call this function, you have to remember the correct order of the arguments. Fortunately, Python allows you to name the arguments when calling the function, as shown below:

```
fancy_print(text, color='yellow',
          background='black', style='bold',
                                justify='left')
fancy_print(text, style='bold',
               color='yellow', justify='left',
                         background='black')
```

As we can see, the order of the arguments does not matter when you use keyword arguments.

When defining the function, it would be a good idea to give defaults. For instance, most of the time, the caller would want left justification, a white background, etc. Using these values as defaults means the caller does not have to specify every single argument every time they call the function.

Here is an example:

```
fancy_print('Student', style='Italic')
```

7.2 Local and Global variables

Let's say we have two functions like the ones below that each use a variable `i`:

```
def fun1():
     for i in range(6):
          print (i)

def fun2():
     i=20
     fun1()
     print(i)
```

A problem that could arise here is that when we call `fun1`, we might mess up the value of `i` in `fun2`. In a large program it would be a nightmare trying to make sure that we don't repeat variable names in different functions, and, fortunately, we don't have to worry about this. When a variable is defined inside a function, it is *local* to that function, which means it essentially does not exist outside that function. This way each function can define its own variables and not have to worry about if those variable names are used in other functions.

On the other hand, sometimes you actually do want the same variable to be available to multiple functions. Such a variable is called a *global* variable. You have to be careful using global variables, especially in larger programs, but a few global variables used judiciously are fine in smaller programs. Here is a short example:

```
def output():
     place = "Cape Town"
     print("%s lives in %s." % (name, place))
     return

place = "Berlin"
name = "Dominic"
```

```
print("%s lives in %s." % (name, place))
output()
```

Output:

```
Dominic lives in Berlin.
Dominic lives in Cape Town.
```

The output consists of these two lines, whereas the first line originates from the main program and the second line from the `print` statement in the function `output()`. At first the two variables `name` and `place` are defined in the main program and printed to stdout. Calling the `output()` function, the variable `place` is locally redefined and `name` comes from the global namespace, instead.

Let's consider another example:

```
def fun1(x):
    x = x + 1

def fun2(L):
    L = L + [4]

p=3
LIST1=[1,2,3]
fun1(p)
fun2(LIST1)
```

When we call `fun1` with `p` and `fun2` with `L`, a question arises: do the functions change the values of `p` and `L`? The answer may surprise you. The value of `p` is unchanged, but the value of `L` is changed. The reason has to do with a difference in the way that Python handles numbers and lists. Lists are said to be *mutable* objects, meaning they can be changed, whereas numbers and strings are *immutable*, meaning they cannot be changed.

If we want to reverse the behavior of the above example so that `p` is modified and `L` is not, do the following:

```
def funl(x):
     x = x + 1
     return x
def fun2(L):
     copy = L[:]
     copy = copy + [1]

p=3
LIST1=[1,2,3]
a=funl(a)
fun2(LIST1)
```

LECTURE 8

PYTHON FOR DATA ANALYSIS

8.1 Data Analysis

We live in the digital era of high technologies, smart devices, and mobile solutions. Data is an essential aspect of any enterprise and business. It's crucial to gather, process, and analyze the data flow and to do that as quickly and accurately as possible. Nowadays, the data volume can be large, which makes information handling time-consuming and expensive. Due to this precise reason, the data science industry is growing at a rapid pace, creating new vacancies and possibilities.

Data analysis is defined as a process of cleaning, transforming, and modeling data to discover useful information for business decision-making. The purpose of Data Analysis is to extract useful information from data and taking the decision based upon the data analysis.

A simple example of Data analysis is whenever we take any decision in our day-to-day life is by thinking about what happened last time or what will happen by choosing that particular decision. This is nothing but analyzing our past or future and making decisions based on it. For that, we gather memories of our past or dreams of our future. So that is nothing but data analysis. Now same thing analyst does for business purposes, is called Data Analysis.

Data analysis tools make it easier for users to process and manipulate data, analyze the relationships and correlations between data sets, and it also helps to identify patterns and trends for interpretation. Here is a complete list of tools used for data analysis in research:

There are several types of Data Analysis techniques that exist based on business and technology. However, the major Data Analysis methods are:

- Text Analysis
- Statistical Analysis
- Diagnostic Analysis
- Predictive Analysis
- Prescriptive Analysis

Text Analysis is also referred to as Data Mining. It is one of the methods of data analysis to discover a pattern in large data sets using databases or data mining tools. It used to transform raw data into business information. Business Intelligence tools are present in the market which is used to take strategic business decisions. Overall it offers a way to extract and examine data and deriving patterns and finally interpretation of the data.

Statistical Analysis shows "What happen?" by using past data in the form of dashboards. Statistical Analysis includes collection, Analysis, interpretation, presentation, and modeling of data. It analyses a set of data or a sample of data.

Diagnostic Analysis shows "Why did it happen?" by finding the cause from the insight found in Statistical Analysis. This Analysis is useful to identify behavior patterns of data. If a new problem arrives in your business process, then you can look into this Analysis to find similar patterns of that problem. And it may have chances to use similar prescriptions for the new problems.

Predictive Analysis shows "what is likely to happen" by using previous data. The simplest data analysis example is like if last year I bought two dresses based on my savings and if this year my salary is increasing double then I can buy four dresses. But of course it's not easy like this because you have to think about other circumstances like

chances of prices of clothes is increased this year or maybe instead of dresses you want to buy a new bike, or you need to buy a house!

So here, this Analysis makes predictions about future outcomes based on current or past data. Forecasting is just an estimate. Its accuracy is based on how much detailed information you have and how much you dig in it.

Prescriptive Analysis combines the insight from all previous Analysis to determine which action to take in a current problem or decision. Most data-driven companies are utilizing Prescriptive Analysis because predictive and descriptive Analysis are not enough to improve data performance. Based on current situations and problems, they analyze the data and make decisions.

The Data Analysis Process is nothing but gathering information by using a proper application or tool which allows you to explore the data and find a pattern in it. Based on that information and data, you can make decisions, or you can get ultimate conclusions.

Data Analysis consists of the following phases:

- Data Requirement Gathering
- Data Collection
- Data Cleaning
- Data Analysis
- Data Interpretation
- Data Visualization
- Data Requirement Gathering

First of all, you have to think about why do you want to do this data analysis? All you need to find out the purpose or aim of doing the Analysis of data. You have to decide which type of data analysis you wanted to do! In this phase, you have to decide what to analyze and how to measure it, you have to understand why you are investigating and what measures you have to use to do this Analysis.

Data Collection. After requirement gathering, you will get a clear idea about what things you have to measure and what should be your findings. Now it's time to collect your data based on requirements. Once you collect your data, remember that the collected

data must be processed or organized for Analysis. As you collected data from various sources, you must have to keep a log with a collection date and source of the data.

Data Cleaning. Now whatever data is collected may not be useful or irrelevant to your aim of Analysis, hence it should be cleaned. The data which is collected may contain duplicate records, white spaces or errors. The data should be cleaned and error free. This phase must be done before Analysis because based on data cleaning, your output of Analysis will be closer to your expected outcome.

Data Analysis. Once the data is collected, cleaned, and processed, it is ready for Analysis. As you manipulate data, you may find you have the exact information you need, or you might need to collect more data. During this phase, you can use data analysis tools and software which will help you to understand, interpret, and derive conclusions based on the requirements.

Data Interpretation. After analyzing your data, it's finally time to interpret your results. You can choose the way to express or communicate your data analysis either you can use simply in words or maybe a table or chart. Then use the results of your data analysis process to decide your best course of action.

Data Visualization. Data visualization is very common in your day to day life; they often appear in the form of charts and graphs. In other words, data shown graphically so that it will be easier for the human brain to understand and process it. Data visualization often used to discover unknown facts and trends. By observing relationships and comparing datasets, you can find a way to find out meaningful information.

8.2 Why Python is widely used for Data Analysis

Python is an increasingly popular tool for data analysis. In recent years, a number of libraries have reached maturity, allowing R and Stata users to take advantage of the beauty, flexibility, and performance of Python without sacrificing the functionality these

older programs have accumulated over the years. Python is a cross-functional, maximally interpreted language that has lots of advantages to offer. The object-oriented programming language is commonly used to streamline large complex data sets. Over and above, having a dynamic semantics plus unmeasured capacities of *RAD (rapid application development),* Python is heavily utilized to script as well. There is one more way to apply Python – as a coupling language.

Another Python's advantage is high readability that helps engineers to save time by typing fewer lines of code for accomplishing the tasks. Being fast, Python jibes well with data analysis. And that's due to heavy support; availability of a whole slew of open-source libraries for different purposes, including but not limited to scientific computing. Therefore, it's not surprising at all that it's claimed to be the preferred programming language for data science. There is a scope of unique features provided that makes Python *a-number-one* option for data analysis. Python is one of the most supported languages nowadays. It has a long list of totally free libraries available for all the users. That's a key factor that gives a strong push for Python at all, and in the data science, too. If you're involved in the field, more than likely, you are acquainted with such names as *Pandas, SciPy, StatsModels*, other libraries that are intensively utilized in the data science community. Noteworthy is that the libraries constantly grow, providing robust solutions. Herewith, you can easily find a solution needed hassle-free without additional expenses.

8.3 Python Data Structures for Data Science

Following are some data structures, which are used in Python. You should be familiar with them in order to use them as appropriate.

Lists are one of the most versatile data structure in Python. A list can simply be defined by writing a list of comma separated values in square brackets. Lists might contain items of different types, but usually the items all have the same type. Python lists are mutable and individual elements of a list can be changed.

Here is a quick example to define a list and then access it:

```
# empty list
my_list = []
# list of integers
my_list = [1, 2, 3]
# list with mixed data types
my_list = [1, "Hello", 3.4]
```

A list can also have another list as an item. This is called a nested list.

```
# nested list
my_list = ["mouse", [8, 4, 6], ['a']]
# List indexing
my_list = ['p', 'r', 'o', 'b', 'e']
# Output: p
print(my_list[0])
# Output: o
print(my_list[2])
# Output: e
print(my_list[4])
# Nested List
n_list = ["Happy", [2, 0, 1, 5]]
# Nested indexing
print(n_list[0][1])
print(n_list[1][3])
# Error! Only integer can be used for indexing
print(my_list[4.0])
```

Output:

```
p
o
e
a
5
Traceback (most recent call last):
```

```
  File "<string>", line 21, in <module>
TypeError: list indices must be integers or
slices, not float
```

Strings can simply be defined by use of single (') , double (") or triple (''') inverted commas. Strings enclosed in tripe quotes (''') can span over multiple lines and are used frequently in docstrings (Python's way of documenting functions). \ is used as an escape character. Please note that Python strings are immutable, so you can not change part of strings.

```
# defining strings in Python
# all of the following are equivalent
my_string = 'Hello'
print(my_string)
my_string = "Hello"
print(my_string)
my_string = '''Hello'''
print(my_string)
# triple quotes string can extend multiple
lines
my_string = """Hello, welcome to
           the world of Python"""
print(my_string)
```

Output:

```
Hello
Hello
Hello
Hello, welcome to
           the world of Python
```

Tuples. A tuple is represented by a number of values separated by commas. Tuples are immutable and the output is surrounded by parentheses so that nested tuples are processed correctly. Additionally, even though tuples are immutable, they can hold mutable data if needed.

Since Tuples are immutable and can not change, they are faster

in processing as compared to lists. Hence, if your list is unlikely to change, you should use tuples, instead of lists.

```
# Different types of tuples
# Empty tuple
my_tuple = ()
print(my_tuple)
# Tuple having integers
my_tuple = (1, 2, 3)
print(my_tuple)
# tuple with mixed datatypes
my_tuple = (1, "Hello", 3.4)
print(my_tuple)
# nested tuple
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)
```

Output:

```
 ()
(1, 2, 3)
(1, 'Hello', 3.4)
('mouse', [8, 4, 6], (1, 2, 3))
```

Dictionary is an unordered set of key: value pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: {}.

```
# empty dictionary
my_dict = {}
# dictionary with integer keys
my_dict = {1: 'apple', 2: 'ball'}
# dictionary with mixed keys
my_dict = {'name': 'John', 1: [2, 4, 3]}
# using dict()
my_dict = dict({1:'apple', 2:'ball'})
# from sequence having each item as a pair
my_dict = dict([(1,'apple'), (2,'ball')])
```

```
# get vs [] for retrieving elements
my_dict = {'name': 'Jack', 'age': 26}
# Output: Jack
print(my_dict['name'])
# Output: 26
print(my_dict.get('age'))
# Trying to access keys which doesn't exist
throws error
# Output None
print(my_dict.get('address'))
# KeyError
print(my_dict['address'])
```

Output:

```
Jack
26
None
Traceback (most recent call last):
  File "<string>", line 15, in <module>
    print(my_dict['address'])
KeyError: 'address'
```

Python Numpy Arrays are a bit like Python lists, but still very much different at the same time. As the name kind of gives away, a NumPy array is a central data structure of the numpy library. The library's name is actually short for "Numeric Python" or "Numerical Python".

Simplest way to create an array in Numpy is to use Python List

```
myPythonList = [1,9,8,3]
```

To convert python list to a numpy array by using the object np.array.

```
numpy_array_from_list = np.array(myPythonList)
```

In practice, there is no need to declare a `Python List`. The operation can be combined.

```
a  = np.array([1,9,8,3])
```

You can also create a `numpy` array from a `Tuple`

You could perform mathematical operations like additions, subtraction, division and multiplication on an array. The syntax is the array name followed by the operation (+.-,*,/) followed by the operand

Example:

```
numpy_array_from_list + 10
```

Output:

```
array([11, 19, 18, 13])
```

This operation adds 10 to each element of the `numpy` array.

You can check the shape of the array with the object shape preceded by the name of the array. In the same way, you can check the type with dtypes.

```
import numpy as np
a  = np.array([1,2,3])
print(a.shape)
print(a.dtype)
```

Output:

```
(3,)
int64
```

An integer is a value without decimal. If you create an array with decimal, then the type will change to float.

```
#### Different type
b  = np.array([1.1,2.0,3.2])
print(b.dtype)
```

Output:

```
float64
```

You can add a dimension with a "," coma
Note that it has to be within the bracket []

```
### 2 dimension
c = np.array([(1,2,3),
              (4,5,6)])
print(c.shape)
```

Output:

```
(2, 3)
```

Series and Dataframe. `Series` is a type of list in Python Pandas which can take integer values, string values, double values and more. But in `Pandas Series` we return an object in the form of list, having index starting from 0 to *n*, Where *n* is the length of values in series. `Pandas DataFrame` is a 2-dimensional labeled data structure with columns of potentially different types. It is generally the most commonly used pandas object. `Series` can only contain single list with index, whereas `dataframe` can be made of more than one series or we can say that a `dataframe` is a collection of series that can be used to analyse the data.

Creating a simple `Series`

```
import pandas as pd
import matplotlib.pyplot as plt
author = ['Jitender', 'Purnima', 'Arpit',
                                  'Jyoti']
auth_series = pd.Series(author)
print(auth_series)
```

Output:

```
0    Jitender
1     Purnima
2       Arpit
3       Jyoti
dtype: object
```

Creating Dataframe from Series

```
import pandas as pd
import matplotlib.pyplot as plt
author = ['Jitender', 'Purnima', 'Arpit',
                                    'Jyoti']
article = [210, 211, 114, 178]
auth_series = pd.Series(author)
article_series = pd.Series(article)
frame = { 'Author': auth_series, 'Article':
                                 article_series }
result = pd.DataFrame(frame)
print(result)
```

Output:

```
     Author  Article
0  Jitender      210
1   Purnima      211
2     Arpit      114
3     Jyoti      178
```

We are combining two series `Author` and `Article` published. Create a dictionary so that we can combine the metadata for series. Metadata is the data of data that can define the series of values. Pass this dictionary to pandas `DataFrame` and finally you can see the result as combination of two series i. e for author and number of articles.

We can add series externally in `dataframe`:

```
import pandas as pd
import matplotlib.pyplot as plt
author = ['Jitender', 'Purnima', 'Arpit',
                                    'Jyoti']
article = [210, 211, 114, 178]
auth_series = pd.Series(author)
article_series = pd.Series(article)
```

```
frame = { 'Author': auth_series, 'Article':
                                 article_series }
result = pd.DataFrame(frame)
age = [21, 21, 24, 23]
result['Age'] = pd.Series(age)
print(result)
```

Output:

```
     Author  Article  Age
0  Jitender      210   21
1   Purnima      211   21
2     Arpit      114   24
3     Jyoti      178   23
```

We have added one more series externally named as age of the authors, then directly added this series in the pandas `dataframe`. Remember one thing if any value is missing then by default it will be converted into `NaN` value i. e null by default.

Pandas DataFrame can be created by passing lists of dictionaries as a input data. By default dictionary keys taken as columns.

```
# Python code demonstrate how to create
# Pandas DataFrame by lists of dicts.
import pandas as pd
# Initialise data to lists.
data = [{'a': 1, 'b': 2, 'c':3},
        {'a':10, 'b': 20, 'c': 30}]
# Creates DataFrame.
df = pd.DataFrame(data)
# Print the data
df
```

Output:

```
           a     b     c
first     NaN   2     3
second   10.0   20    30
```

8.4 Python Libraries for Data Science

Following are a list of libraries, you will need for any scientific computations and data analysis:

- **NumPy** stands for Numerical Python. The most powerful feature of NumPy is n-dimensional array. This library also contains basic linear algebra functions, Fourier transforms, advanced random number capabilities and tools for integration with other low level languages like Fortran, C and C++.
- **SciPy** stands for Scientific Python. SciPy is built on NumPy. It is one of the most useful library for variety of high level science and engineering modules like discrete Fourier transform, Linear Algebra, Optimization and Sparse matrices.
- **Matplotlib** for plotting vast variety of graphs, starting from histograms to line plots to heat plots. You can use Pylab feature in ipython notebook (ipython notebook –pylab = inline) to use these plotting features inline. If you ignore the inline option, then pylab converts ipython environment to an environment, very similar to Matlab. You can also use Latex commands to add math to your plot.
- **Pandas** for structured data operations and manipulations. It is extensively used for data munging and preparation. Pandas were added relatively recently to Python and have been instrumental in boosting Python's usage in data scientist community.
- **Scikit Learn** for machine learning. Built on NumPy, SciPy and matplotlib, this library contains a lot of efficient tools for machine learning and statistical modeling including classification, regression, clustering and dimensionality reduction.

- **Statsmodels** for statistical modeling. Statsmodels is a Python module that allows users to explore data, estimate statistical models, and perform statistical tests. An extensive list of descriptive statistics, statistical tests, plotting functions, and result statistics are available for different types of data and each estimator.
- **Seaborn** for statistical data visualization. Seaborn is a library for making attractive and informative statistical graphics in Python. It is based on matplotlib. Seaborn aims to make visualization a central part of exploring and understanding data.
- **Bokeh** for creating interactive plots, dashboards and data applications on modern web-browsers. It empowers the user to generate elegant and concise graphics in the style of D3.js. Moreover, it has the capability of high-performance interactivity over very large or streaming datasets.
- **Blaze** for extending the capability of Numpy and Pandas to distributed and streaming datasets. It can be used to access data from a multitude of sources including Bcolz, MongoDB, SQLAlchemy, Apache Spark, PyTables, etc. Together with Bokeh, Blaze can act as a very powerful tool for creating effective visualizations and dashboards on huge chunks of data.
- **Scrapy** for web crawling. It is a very useful framework for getting specific patterns of data. It has the capability to start at a website home url and then dig through web-pages within the website to gather information.

LECTURE 9

DATA EXPLORATION IN PYTHON: USING PANDAS

Data Exploration – finding out more about the data we have. In order to explore our data further, let's use Pandas. Pandas is one of the most useful data analysis library in Python. They have been instrumental in increasing the use of Python in data science community. We will now use Pandas to read a data set from an *Analytics Vidhya competition*, perform exploratory analysis and build our first basic categorization algorithm for solving this problem. Before loading the data, lets understand the 2 key data structures in Pandas – Series and DataFrames

Series can be understood as a 1 dimensional labelled/indexed array. You can access individual elements of this series through these labels.

A dataframe is similar to Excel workbook – you have column names referring to columns and you have rows, which can be accessed with use of row numbers. The essential difference being that column names and row numbers are known as column and row index, in case of dataframes.

Series and dataframes form the core data model for Pandas in Python. The data sets are first to read into these dataframes and then various operations (e. g. group by, aggregation etc.) can be applied very easily to its columns.

Let's use dataset which has the following description of the variables:

Variable	Description
Loan_ID	Unique Loan ID
Gender	Male/Female
Married	Applicant married (Y/N)
Dependents	Number of dependents
Education	Applicant Education

Variable	Description
	(Graduate/ Under Graduate)
Self_Employed	Self employed (Y/N)
ApplicantIncome	Applicant income
CoapplicantIncome	Coapplicant income
LoanAmount	Loan amount in thousands
Loan_Amount_Term	Term of loan in months
Credit_History	Credit history meets guidelines
Property_Area	Urban/ Semi Urban/ Rural
Loan_Status	Loan approved (Y/N)

After importing the library (file `train.csv`), you read the dataset using function `read_csv()`. This is how the code looks like till this stage:

```
import pandas as pd
import numpy as np
import matplotlib as plt
#Reading the dataset in a dataframe using Pandas
df = pd.read_csv("train.csv")
```

Once you have read the dataset, you can have a look at few top rows by using the function `head()`

```
In [3]: df.head(10)  #Printing first 10 rows of dataset
```

Out[3]:

	Loan_ID	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Cr
0	LP001002	Male	No	0	Graduate	No	5849	0	NaN	360	1
1	LP001003	Male	Yes	1	Graduate	No	4583	1508	128	360	1
2	LP001005	Male	Yes	0	Graduate	Yes	3000	0	66	360	1
3	LP001006	Male	Yes	0	Not Graduate	No	2583	2358	120	360	1
4	LP001008	Male	No	0	Graduate	No	6000	0	141	360	1
5	LP001011	Male	Yes	2	Graduate	Yes	5417	4196	267	360	1
6	LP001013	Male	Yes	0	Not Graduate	No	2333	1516	95	360	1
7	LP001014	Male	Yes	3+	Graduate	No	3036	2504	158	360	0
8	LP001018	Male	Yes	2	Graduate	No	4006	1526	168	360	1
9	LP001020	Male	Yes	1	Graduate	No	12841	10968	349	360	1

```
df.head(10)
```

This should print 10 rows. Alternately, you can also look at more rows by printing the dataset. Next, you can look at summary of numerical fields by using `describe()` function

```
df.describe()
```

```
In [4]: df.describe()  #Get summary of numerical variables
```

Out[4]:

	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History
count	614.000000	614.000000	592.000000	600.00000	564.000000
mean	5403.459283	1621.245798	146.412162	342.00000	0.842199
std	6109.041673	2926.248369	85.587325	65.12041	0.364878
min	150.000000	0.000000	9.000000	12.00000	0.000000
25%	2877.500000	0.000000	100.000000	360.00000	1.000000
50%	3812.500000	1188.500000	128.000000	360.00000	1.000000
75%	5795.000000	2297.250000	168.000000	360.00000	1.000000
max	81000.000000	41667.000000	700.000000	480.00000	1.000000

As we can see, `describe()` function would provide `count`, `mean`, `standard deviation` (`std`), `min`, `quartiles` and `max` in its output. Let's remember what this basic statistics mean:

Mean. This is most common used measure of central tendency. The mean is simply the average value of a data set. We have all averaged a set of exam grades before. That is an example of mean.

Standard Deviation is a measure of variation that is the simply the square root of the variance.

Variance is a measure of variation. That is it describes how the data is distributed about the mean.

Quartiles are the values that divide a list of numbers into quarters. To calculate Quartiles put the list of numbers in order, then cut the list into four equal parts. The Quartiles are at the "cuts". Like

this:

Example: 5, 7, 4, 4, 6, 2, 8

Put them in order: 2, 4, 4, 5, 6, 7, 8
Cut the list into quarters:

And the result is:
Quartile 1 (Q1) = 4
Quartile 2 (Q2), which is also the Median, = 5
Quartile 3 (Q3) = 7

If we look at the output of `describe()` function, we can see:
1. `LoanAmount` has (614 – 592) 22 missing values;
2. `Loan_Amount_Term` has (614 – 600) 14 missing values;
3. `Credit_History` has (614 – 564) 50 missing values;

We can also look that about 84 % applicants have a credit history. How? The mean of `Credit_History` field is 0.84 (Remember, `Credit_History` has value 1 for those who have a credit history and 0 otherwise)

The `ApplicantIncome` distribution seems to be in line with expectation. Same with `CoapplicantIncome`

```
df['Property_Area'].value_counts()
```

Similarly, we can look at unique values of port of credit history. Note that `dfname['column_name']` is a basic indexing technique to acess a particular column of the dataframe. It can be a list of columns as well.

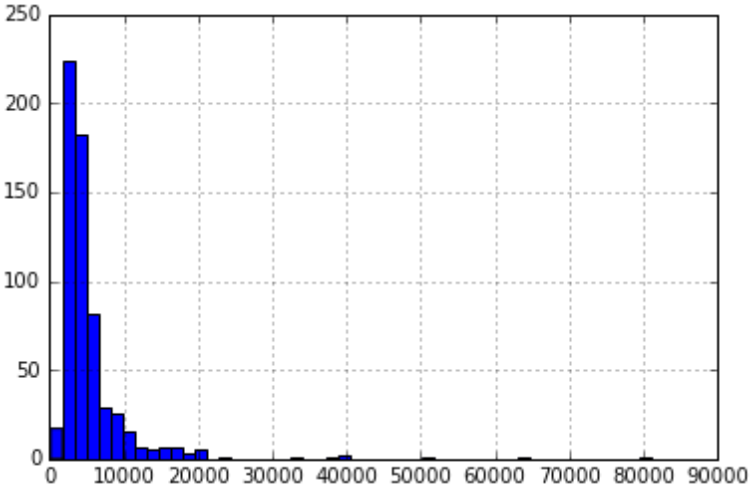
9.1 Distribution analysis

Now that we are familiar with basic data characteristics, let us study distribution of various variables. Let us start with numeric variables –

namely `ApplicantIncome` and `LoanAmount`.

Let's start by plotting the histogram of `ApplicantIncome` using the following commands:

```
df['ApplicantIncome'].hist(bins=50)
```



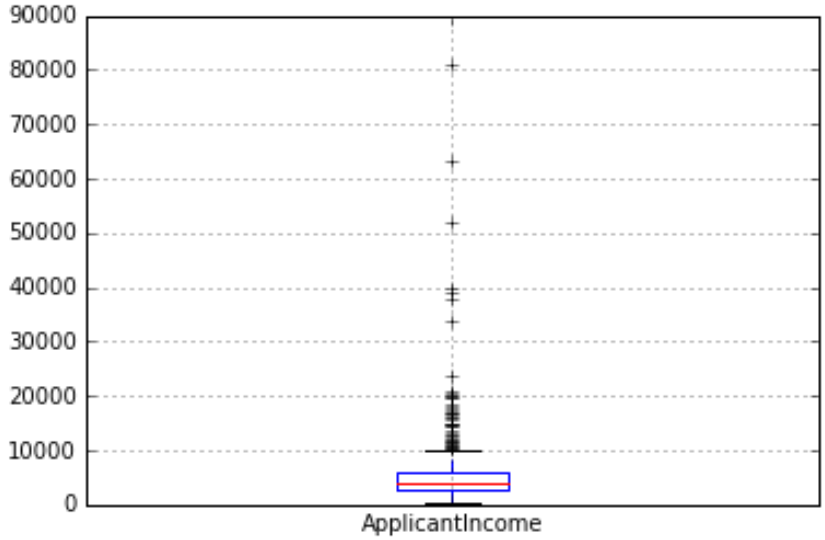
Here we observe that there are few extreme values. This is also the reason why 50 bins are required to depict the distribution clearly.

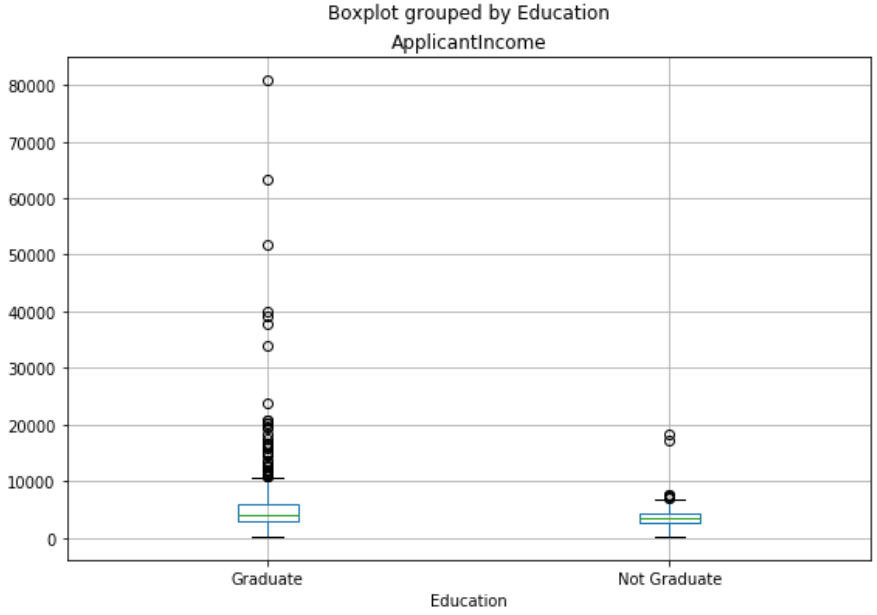
Next, we look at box plots to understand the distributions. Box plot for fare can be plotted by:

```
df.boxplot(column='ApplicantIncome')
```

This confirms the presence of a lot of outliers/extreme values. This can be attributed to the income disparity in the society. Part of this can be driven by the fact that we are looking at people with different education levels. Let us segregate them by `Education`:

```
df.boxplot(column='ApplicantIncome', by =
                                    'Education')
```

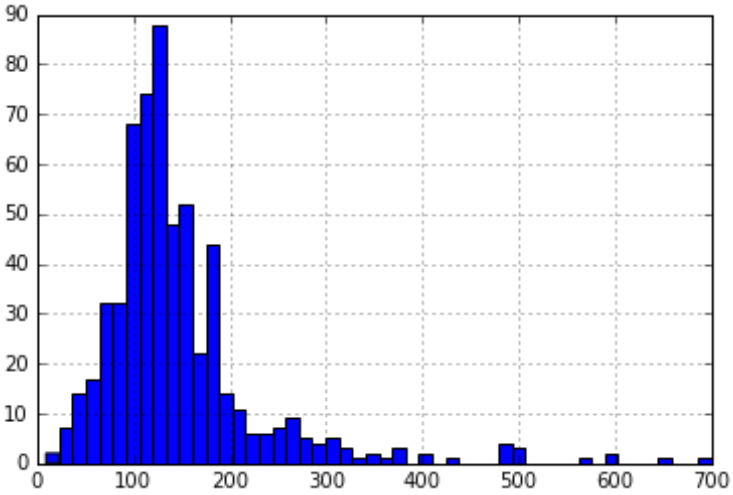

90000
80000
70000
60000
50000
40000
30000
20000
10000
0
ApplicantIncome


Boxplot grouped by Education
ApplicantIncome
80000
70000
60000
50000
40000
30000
20000
10000
0
Graduate
Not Graduate
Education

We can see that there is no substantial different between the mean income of graduate and non-graduates. But there are a higher number of graduates with very high incomes, which are appearing to be the outliers.

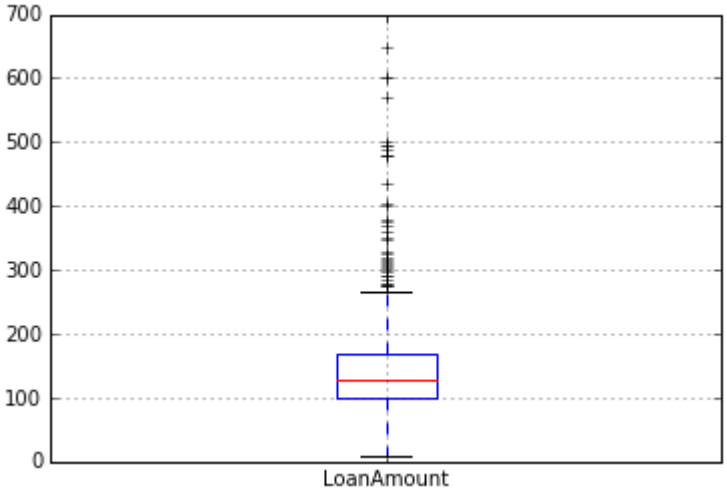
Now, Let's look at the histogram and boxplot of `LoanAmount` using the following command:

```
df['LoanAmount'].hist(bins=50)
```




```
df.boxplot(column='LoanAmount')
```

Again, there are some extreme values. Clearly, both `ApplicantIncome` and `LoanAmount` require some amount of data munging. `LoanAmount` has missing and well as extreme values values, while `ApplicantIncome` has a few extreme values, which demand deeper understanding. We will take this up in coming subsection.



9.2 Categorical variable analysis

Now that we understand distributions for `ApplicantIncome` and `LoanIncome`, let us understand categorical variables in more details. For instance, let us look at the chances of getting a loan based on credit history. Let us use the function `pivot_table` (from `Pandas`) to do this. `Pivot tables` are one of `Excel's` most powerful features. A `pivot table` is a table of statistics that summarizes the data of a more extensive table (such as from a database, spreadsheet, or business intelligence program). This summary might include sums, averages, or other statistics, which the pivot table groups together in a meaningful way. `Pivot tables` are a technique in data processing. They arrange and rearrange (or "pivot") statistics in order to draw attention to useful information.

`Pandas` provides a similar function called `pivot_table()`. `Pandas pivot_table()` is a simple function but can produce very powerful analysis very quickly.

```
temp1=df['Credit_History'].value_counts(
                                ascending=True)
```

```
temp2=df.pivot_table(values='Loan_Status',
index=['Credit_History'],aggfunc=lambda x:
                x.map({'Y':1,'N':0}).mean())
# Loan status is coded as 1 for Yes and 0 for
# No. So the mean represents the probability
# of getting loan.
print ('Frequency Table for Credit History:')
print (temp1)
print ('\nProbility of getting loan for each
                    Credit History class:')
print (temp2)
```

Output:

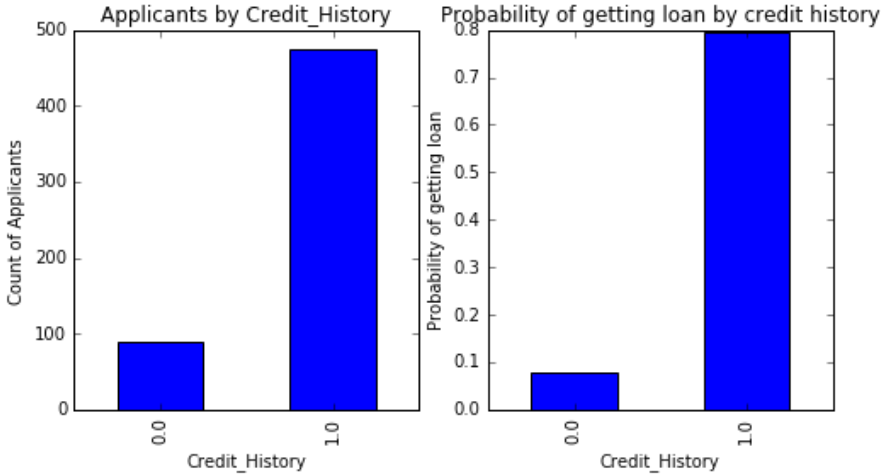
```
Frequency Table for Credit History:
0     89
1    475
Name: Credit_History, dtype: int64

Probility of getting loan for each Credit History class:
Credit_History
0    0.078652
1    0.795789
Name: Loan_Status, dtype: float64
```

This can be plotted as a bar chart using the “matplotlib” library with following code:

```
import matplotlib.pyplot as plt fig =
plt.figure(figsize=(8,4))
ax1=fig.add_subplot(121)
ax1.set_xlabel('Credit_History')
ax1.set_ylabel('Count of Applicants')
ax1.set_title("Applicants by Credit_History")
temp1.plot(kind='bar')
ax2 = fig.add_subplot(122)
temp2.plot(kind = 'bar')
ax2.set_xlabel('Credit_History')
```

```
ax2.set_ylabel('Probability of getting loan')
ax2.set_title("Probability of getting loan by
                                credit history")
```



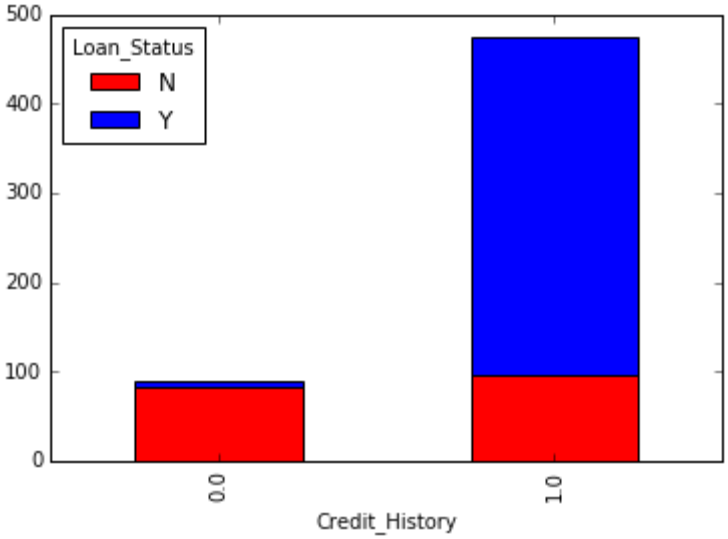
This shows that the chances of getting a loan are eight-fold if the applicant has a valid credit history. You can plot similar graphs by `Married`, `Self-Employed`, `Property_Area`, etc.

Alternately, these two plots can also be visualized by combining them in a stacked chart:

```
temp3 = pd.crosstab(df['Credit_History'],
                            df['Loan_Status'])
temp3.plot(kind='bar', stacked=True,
           color=['red','blue'], grid=False)
```

We have just created two basic classification algorithms here, one based on credit history, while other on 2 categorical variables.

We just saw how we can do exploratory analysis in Python using Pandas. Next let's explore `ApplicantIncome` and `LoanStatus` variables further, perform data munging and create a dataset for applying various modeling techniques.



9.3 Data Munging in Python

While our exploration of the data, we found a few problems in the data set, which needs to be solved before the data is ready for a good model. Here are the problems, we are already aware of:

1. There are missing values in some variables. We should estimate those values wisely depending on the amount of missing values and the expected importance of variables.
2. While looking at the distributions, we saw that `ApplicantIncome` and `LoanAmount` seemed to contain extreme values at either end. Though they might make intuitive sense, but should be treated appropriately.

In addition to these problems with numerical fields, we should also look at the non-numerical fields i. e. `Gender`, `Property_Area`, `Married`, `Education` and `Dependents` to see, if they contain any useful information.

Let us look at missing values in all the variables because most of the models don't work with missing data and even if they do, imputing them helps more often than not. So, let us check the number of nulls / `NaNs` in the dataset.

```
df.apply(lambda x: sum(x.isnull()),axis=0)
```

This command should tell us the number of missing values in each column as `isnull()` returns 1, if the value is null.

```
In [14]: df.apply(lambda x: sum(x.isnull()),axis=0)

Out[14]: Loan_ID              0
         Gender              13
         Married              3
         Dependents          15
         Education            0
         Self_Employed       32
         ApplicantIncome      0
         CoapplicantIncome    0
         LoanAmount          22
         Loan_Amount_Term    14
         Credit_History      50
         Property_Area        0
         Loan_Status          0
         dtype: int64
```

Though the missing values are not very high in number, but many variables have them and each one of these should be estimated and added in the data.

It should be noted that missing values may not always be `NaNs`. For instance, if the `Loan_Amount_Term` is 0, does it makes sense or would you consider that missing? I suppose your answer is missing and you're right. So we should check for values which are unpractical.

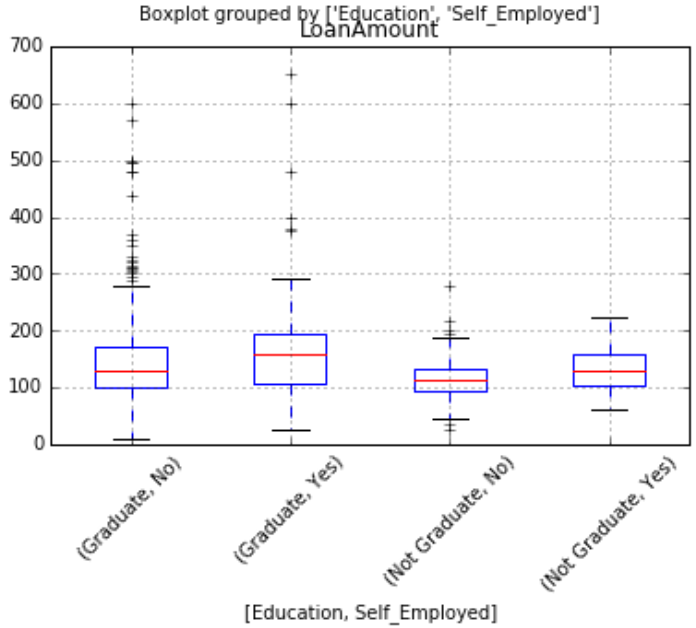
There are numerous ways to fill the missing values of loan amount – the simplest being replacement by mean, which can be done by following code:

```
df['LoanAmount'].fillna(df['LoanAmount'].
                        mean(), inplace=True)
```

The other extreme could be to build a supervised learning model to predict loan amount on the basis of other variables and then use age along with other variables to predict survival.

Since, the purpose now is to bring out the steps in data munging, let's rather take an approach, which lies somewhere in between these 2 extremes. A key hypothesis is that the whether a person is educated or self-employed can combine to give a good estimate of loan amount.

First, let's look at the boxplot to see if a trend exists:



Thus we see some variations in the median of loan amount for each group and this can be used to impute the values. But first, we have to ensure that each of `Self_Employed` and `Education` variables should not have a missing values.

As we say earlier, `Self_Employed` has some missing values. Let's look at the frequency table:

```
In [40]: df['Self_Employed'].value_counts()

Out[40]: No     500
         Yes     82
         Name: Self_Employed, dtype: int64
```

Since ~ 86 % values are "No", it is safe to impute the missing values as "No" as there is a high probability of success. This can be done using the following code:

```
df['Self_Employed'].fillna('No',inplace=True)
```

Now, we will create a `Pivot table`, which provides us median values for all the groups of unique values of `Self_Employed` and `Education` features. Next, we define a function, which returns the values of these cells and apply it to fill the missing values of loan amount:

```
table = df.pivot_table(values='LoanAmount',
    index='Self_Employed',columns='Education',
                           aggfunc=np.median)
# Define function to return value of this
# pivot_table
def fage(x):
     return table.loc[x['Self_Employed'],
                              x['Education']]

# Replace missing values
df['LoanAmount'].fillna(df[df['LoanAmount'].
```

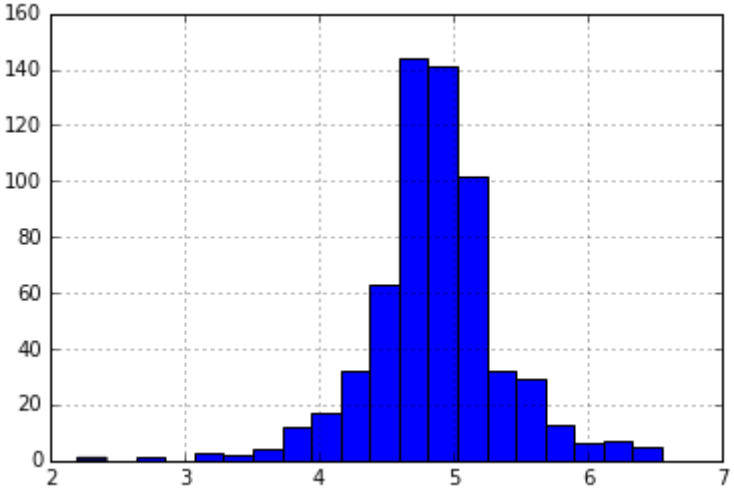
```
        isnull()].apply(fage, axis=1),
                                inplace=True)
```

This should provide you a good way to impute missing values of loan amount. It should be noted that ehis method will work only if you have not filled the missing values in `Loan_Amount` variable using the previous approach, i. e. using mean.

Let's analyze `Loan_Amount` first. Since the extreme values are practically possible, i. e. some people might apply for high value loans due to specific needs. So instead of treating them as outliers, let's try a log transformation to nullify their effect:

```
df['LoanAmount_log']=np.log(df['LoanAmount'])
df['LoanAmount_log'].hist(bins=20)
```

Looking at the histogram again:

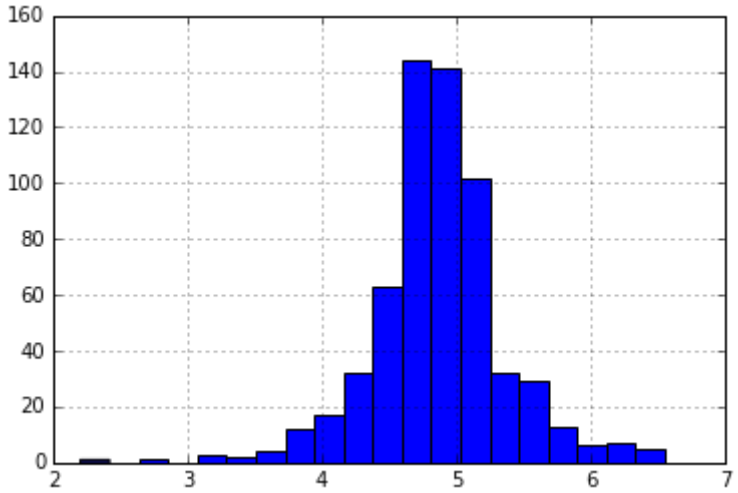


Now the distribution looks much closer to normal and effect of extreme values has been significantly subsided.

Coming to `ApplicantIncome`. One intuition can be that some applicants have lower income but strong support Co-applicants. So it might be a good idea to combine both incomes as total income

and take a log transformation of the same.

```
df['TotalIncome'] = df['ApplicantIncome'] +
                    df['CoapplicantIncome']
df['TotalIncome_log']=
                    np.log(df['TotalIncome'])
df['LoanAmount_log'].hist(bins=20)
```



Now we see that the distribution is much better than before. Next, we will look at making predictive models.

LECTURE 10

BUILDING A PREDICTIVE MODEL IN PYTHON

After, we have made the data useful for modeling, let's now look at the python code to create a predictive model on our data set. Skicit-Learn (sklearn) is the most commonly used library in Python for this purpose and we will follow the trail.

Since, sklearn requires all inputs to be numeric, we should convert all our categorical variables into numeric by encoding the categories. Before that we will fill all the missing values in the dataset. This can be done using the following code:

```
df['Gender'].fillna(df['Gender'].mode()[0],
                                        inplace=True)
df['Married'].fillna(df['Married'].mode()[0],
                                        inplace=True)
df['Dependents'].fillna(df['Dependents'].mode()[0]
                                    , inplace=True)
df['Loan_Amount_Term'].
     fillna(df['Loan_Amount_Term'].mode()[0],
                                        inplace=True)
df['Credit_History'].fillna(df['Credit_History
          '].mode()[0], inplace=True)
from sklearn.preprocessing import LabelEncoder
var_mod=['Gender','Married','Dependents',
     'Education','Self_Employed','Property_Area',
                                    'Loan_Status']
le = LabelEncoder()
for i in var_mod:
    df[i] = le.fit_transform(df[i])
df.dtypes
```

Next, we will import the required modules. Then we will define a generic classification function, which takes a model as input and determines the Accuracy and Cross-Validation scores.

```
#Import models from scikit learn module:
from sklearn.linear_model import
                            LogisticRegression
#For K-fold cross validation
from sklearn.cross_validation import KFold
from sklearn.ensemble import
                       RandomForestClassifier
from sklearn.tree import
     DecisionTreeClassifier, export_graphviz
from sklearn import metrics
# Generic function for making a classification
# model and accessing performance:

def classification_model(model, data,
                             predictors, outcome):
  #Fit the model:
  model.fit(data[predictors],data[outcome])
  #Make predictions on training set:
  predictions=model.predict(data[predictors])
  #Print accuracy
  accuracy=metrics.accuracy_score(predictions,
                                  data[outcome])
  print("Accuracy:%s"%"{0:.3%}".format(accuracy))
  #Perform k-fold cross-validation with 5 folds
  kf = KFold(data.shape[0], n_folds=5)
  error = []

  for train, test in kf:
    # Filter training data
    train_predictors
                =(data[predictors].iloc[train,:])
    # The target we're using to train the
                                       algorithm.
    train_target = data[outcome].iloc[train]
    # Training the algorithm using the predictors
                                       and target.
    model.fit(train_predictors, train_target)
    #Record error from each cross-validation run
    error.append(model.score(data[predictors].iloc
```

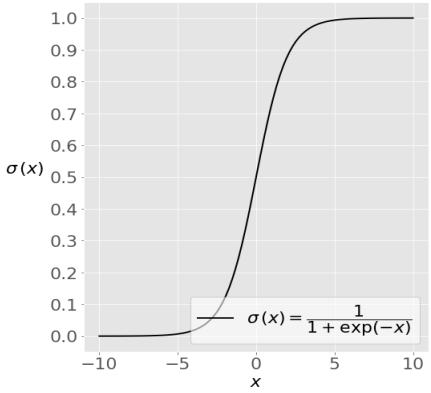
```
            [test,:], data[outcome].iloc[test]))
    print ("Cross-Validation Score : %s" %
                  "{0:.3%}".format(np.mean(error)))
    # Fit the model again so that it can be refered
    # outside the function:
    model.fit(data[predictors],data[outcome])
```

10.1 Logistic Regression

Let's make our first Logistic Regression model. One way would be to take all the variables into the model but this might result in overfitting. In simple words, taking all variables might result in the model understanding complex relations specific to the data and will not generalize well. Logistic regression is a fundamental classification technique. It belongs to the group of linear classifiers and is somewhat similar to polynomial and **linear regression**. Logistic regression is fast and relatively uncomplicated, and it's convenient for you to interpret the results. Although it's essentially a method for binary classification, it can also be applied to multiclass problems.

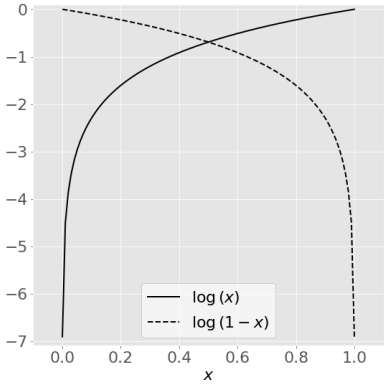
You'll need an understanding of the sigmoid function and the natural logarithm function to understand what logistic regression is and how it works.

This image shows the sigmoid function (or S-shaped curve) of some variable x:



The sigmoid function has values very close to either 0 or 1 across most of its domain. This fact makes it suitable for application in classification methods.

The next image depicts the natural logarithm $\log(x)$ of some variable x, for values of x between 0 and 1:



When you're implementing the logistic regression of some dependent variable y on the set of independent variables $\mathbf{x} = (x_1, ..., x_r)$, where r is the number of predictors (or inputs), you start with the known values of the predictors $\mathbf{x}_i$ and the corresponding actual response (or output) y_i for each observation $i = 1, \ldots, n$.

Our goal is to find the **logistic regression function** $p(\mathbf{x})$ such that the **predicted responses** $p(\mathbf{x}_i)$ are as close as possible to the **actual response** y_i for each observation $i = 1, \ldots, n$. Remember that the actual response can be only 0 or 1 in binary classification problems! This means that each $(\mathbf{x}_i)$ should be close to either 0 or 1. That's why it's convenient to use the sigmoid function. Once you have the logistic regression function $(\mathbf{x})$, you can use it to predict the outputs for new and unseen inputs, assuming that the underlying mathematical dependence is unchanged.

Logistic regression is a linear classifier, so you'll use a linear function $(\mathbf{x}) = b_0 + b_1x_1 + \cdots + b_rx_r$, also called the **logit**. The variables

b_0, b_1, …, b_r are the **estimators** of the regression coefficients, which are also called the **predicted weights** or just **coefficients**.

The logistic regression function is the sigmoid function of **x**: (**x**) = 1 / (1 + exp(−f(**x**)). As such, it's often close to either 0 or 1. The function (**x**) is often interpreted as the predicted probability that the output for a given **x** is equal to 1. Therefore, 1 − (x) is the probability that the output is 0.

Logistic regression determines the best predicted weights b_0, b_1, …, b_r such that the function p(**x**) is as close as possible to all actual responses y_i, i = 1, …, n, where n is the number of observations. The process of calculating the best weights using available observations is called **model training** or **fitting**.

To get the best weights, you usually maximize the **log-likelihood function (LLF)** for all observations i = 1, …, n. This method is called the **maximum likelihood estimation** and is represented by the equation

$$\text{LLF} = \Sigma_i(y_i \log(p(\mathbf{x}_i)) + (1 - y_i) \log(1 - p(\mathbf{x}_i))).$$

When y_i = 0, the LLF for the corresponding observation is equal to log(1 − p(**x**$_i$)). If (**x**$_i$) is close to y_i = 0, then log(1 − p(**x**$_i$)) is close to 0. This is the result you want. If (**x**$_i$) is far from 0, then log(1 − p(**x**$_i$)) drops significantly. You don't want that result because your goal is to obtain the maximum LLF. Similarly, when y_i = 1, the LLF for that observation is y_i log(p(**x**$_i$)). If (**x**$_i$) is close to y_i = 1, then log(p(**x**$_i$)) is close to 0. If (**x**$_i$) is far from 1, then log(p(**x**$_i$)) is a large negative number.

There are several mathematical approaches that will calculate the best weights that correspond to the maximum LLF, but that's beyond the scope of this Lecture notes. For now, you can leave these details to the logistic regression Python libraries you'll learn to use here!

Once you determine the best weights that define the function (**x**), you can get the predicted outputs (**x**$_i$) for any given input **x**$_i$. For each observation i = 1, …, n, the predicted output is 1 if p(**x**$_i$) > 0.5 and 0 otherwise. The threshold doesn't have to be 0.5, but it usually

is. You might define a lower or higher value if that's more convenient for your situation.

There's one more important relationship between (**x**) and (**x**), which is that $\log(p(\mathbf{x}) / (1 - p(\mathbf{x}))) = f(\mathbf{x})$. This equality explains why (**x**) is the **logit**. It implies that (**x**) = 0.5 when (**x**) = 0 and that the predicted output is 1 if (**x**) > 0 and 0 otherwise.

Binary classification has four possible types of results:

- **True negatives:** correctly predicted negatives (zeros).
- **True positives:** correctly predicted positives (ones).
- **False negatives:** incorrectly predicted negatives (zeros).
- **False positives:** incorrectly predicted positives (ones).

You usually evaluate the performance of your classifier by comparing the actual and predicted outputsand counting the correct and incorrect predictions.

The most straightforward indicator of **classification accuracy** is the ratio of the number of correct predictions to the total number of predictions (or observations). Other indicators of binary classifiers include the following:

- **The** positive predictive value is the ratio of the number of true positives to the sum of the numbers of true and false positives.
- **The** negative predictive value is the ratio of the number of true negatives to the sum of the numbers of true and false negatives.
- **The** sensitivity (also known as recall or true positive rate) is the ratio of the number of true positives to the number of actual positives.
- **The** specificity (or true negative rate) is the ratio of the number of true negatives to the number of actual negatives.

The most suitable indicator depends on the problem of interest. In this tutorial, you'll use the most straightforward form of classification accuracy.

Let's return back to the data set from an *Analytics Vidhya competition.* We can easily make some intuitive hypothesis to set the ball rolling. The chances of getting a loan will be higher for:

- Applicants having a credit history (remember we observed this in exploration?).
- Applicants with higher applicant and co-applicant incomes.
- Applicants with higher education level.
- Properties in urban areas with high growth perspectives.

So let's make our first model with 'Credit_History'.

```
outcome_var = 'Loan_Status'
model = LogisticRegression()
predictor_var = ['Credit_History']
classification_model(model, df,predictor_var,
                                        outcome_var)
```

Output:

```
Accuracy : 80.945% Cross-Validation Score :
80.946%
```

Generally we expect the accuracy to increase on adding variables. But this is a more challenging case. The accuracy and cross-validation score are not getting impacted by less important variables. Credit_History is dominating the mode. We have two options now:

- Feature Engineering: derive new information and try to predict those.
- Better modeling techniques. Let's explore this next.

10.2 Decision Tree

Decision tree is another method for making a predictive model. It is known to provide higher accuracy than logistic regression model.

Decision Tree algorithm belongs to the family of supervised learning algorithms. Unlike other supervised learning algorithms, the decision tree algorithm can be used for solving **regression and classification problems** too.

The goal of using a Decision Tree is to create a training model that can use to predict the class or value of the target variable by

learning simple decision rules inferred from prior data (training data).

In Decision Trees, for predicting a class label for a record we start from the **root** of the tree. We compare the values of the root attribute with the record's attribute. On the basis of comparison, we follow the branch corresponding to that value and jump to the next node.

Types of decision trees are based on the type of target variable we have. It can be of two types:

- **Categorical Variable Decision Tree:** Decision Tree which has a categorical target variable then it called a **Categorical variable decision tree.**
- **Continuous Variable Decision Tree:** Decision Tree has a continuous target variable then it is called **Continuous Variable Decision Tree.**

Let's say we have a problem to predict whether a customer will pay his renewal premium with an insurance company (yes/ no). Here we know that the income of customers is a significant variable but the insurance company does not have income details for all customers. Now, as we know this is an important variable, then we can build a decision tree to predict customer income based on occupation, product, and various other variables. In this case, we are predicting values for the continuous variables.

Important Terminology related to Decision Trees:

- **Root Node:** It represents the entire population or sample and this further gets divided into two or more homogeneous sets.
- **Splitting:** It is a process of dividing a node into two or more sub-nodes.
- **Decision Node:** When a sub-node splits into further sub-nodes, then it is called the decision node.
- **Leaf/Terminal Node:** Nodes do not split is called Leaf or Terminal node.
- **Pruning:** When we remove sub-nodes of a decision node, this process is called pruning. You can say the opposite process of splitting.

- **Branch/Sub-Tree:** A subsection of the entire tree is called branch or sub-tree.
- **Parent and Child Node:** A node, which is divided into sub-nodes is called a parent node of sub-nodes whereas sub-nodes are the child of a parent node.

Decision trees classify the examples by sorting them down the tree from the root to some leaf/terminal node, with the leaf/terminal node providing the classification of the example. Each node in the tree acts as a test case for some attribute, and each edge descending from the node corresponds to the possible answers to the test case. This process is recursive in nature and is repeated for every subtree rooted at the new node. The decision of making strategic splits heavily affects a tree's accuracy. The decision criteria are different for classification and regression trees.

Decision trees use multiple algorithms to decide to split a node into two or more sub-nodes. The creation of sub-nodes increases the homogeneity of resultant sub-nodes. In other words, we can say that the purity of the node increases with respect to the target variable. The decision tree splits the nodes on all available variables and then selects the split which results in most homogeneous sub-nodes.

Let us look at some **ID3** algorithm used in Decision Trees. It begins with the original set *S* as the root node. On each iteration of the algorithm, it iterates through the very unused attribute of the set S and calculates **Entropy** and **Information gain (IG)** of this attribute. It then selects the attribute which has the smallest Entropy or Largest Information gain. The set *S* is then split by the selected attribute to produce a subset of the data. The algorithm continues to recur on each subset, considering only attributes never selected before. If the dataset consists of ***N*** attributes then deciding which attribute to place at the root or at different levels of the tree as internal nodes is a complicated step. By just randomly selecting any node to be the root can't solve the issue. If we follow a random approach, it may give us bad results with low accuracy.

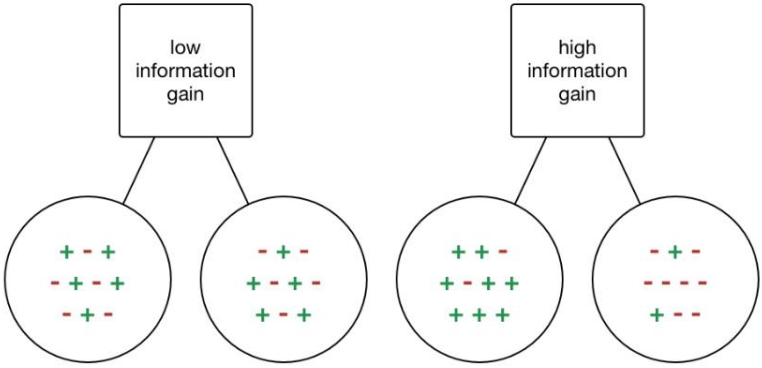
Let's consider an algorithm of building of Decision Tree based on Entropy. Entropy is a measure of the randomness in the information being processed. The higher the entropy, the harder it is to draw any

conclusions from that information. **ID3 follows the rule — a branch with an entropy of zero is a leaf node and a branch with entropy more than zero needs further splitting.**
Mathematically Entropy for 1 attribute is represented as:

$$\text{Entropy}(S) = -\Sigma_i\, p_i \log(p_i),$$

where S is Current state, $p_i \rightarrow$ Probability of an event i of state S or Percentage of class i in a node of state S.

Information gain or **IG** is a statistical property that measures how well a given attribute separates the training examples according to their target classification. Constructing a decision tree is all about finding an attribute that returns the highest information gain and the smallest entropy.



Information gain is a decrease in entropy. It computes the difference between entropy before split and average entropy after split of the dataset based on given attribute values.

Let's return back to the data set from an *Analytics Vidhya competition* and make our model:

```
model = DecisionTreeClassifier()
predictor_var=['Credit_History','Gender','Married'
                                        ,'Education']
classification_model(model,df,predictor_var,
```

```
                                                    outcome_var)
```

Output:

```
Accuracy : 81.930% Cross-Validation Score : 76.656%
```

Here the model based on categorical variables is unable to have an impact because Credit History is dominating over them. Let's try a few numerical variables:

```
predictor_var =
['Credit_History','Loan_Amount_Term','LoanAmou
nt_log']
classification_model(model,
df,predictor_var,outcome_var)
```

Output:

```
Accuracy : 92.345% Cross-Validation Score : 71.009%
```

Here we observed that although the accuracy went up on adding variables, the cross-validation error went down. This is the result of model over-fitting the data. Let's try an even more sophisticated algorithm and see if it helps:

10.3 Random Forest

The common problem with Decision trees, especially having a table full of columns, they fit a lot. Sometimes it looks like the tree memorized the training data set. If there is no limit set on a decision tree, it will give you 100 % accuracy on the training data set because in the worst case it will end up making 1 leaf for each observation. Thus this affects the accuracy when predicting samples that are not part of the training set. One of the ways to remove overfitting is Random Forest. Random forest is another algorithm for solving the classification problem. An advantage with Random Forest is that we can make it work with all the features and it returns a feature importance matrix which can be used to select features.

Two key concepts that give it the name random:

- A random sampling of training data set when building trees.
- Random subsets of features considered when splitting nodes.

A technique known as bagging is used to create an ensemble of trees where multiple training sets are generated with replacement. In the bagging technique, a data set is divided into N samples using randomized sampling. Then, using a single learning algorithm a model is built on all samples. Later, the resultant predictions are combined using voting or averaging in parallel.

```
model=RandomForestClassifier(n_estimators=100)
predictor_var=['Gender', 'Married',
                    'Dependents', 'Education',
          'Self_Employed', 'Loan_Amount_Term',
          'Credit_History', 'Property_Area',
        'LoanAmount_log','TotalIncome_log']
classification_model(model,df,predictor_var,
                                    outcome_var)
```

Output:

```
Accuracy : 100.000% Cross-Validation Score :
78.179%
```

Here we see that the accuracy is 100 % for the training set. This is the ultimate case of overfitting and can be resolved in two ways:

- Reducing the number of predictors.
- Tuning the model parameters.

Let's try both of these. First we see the feature importance matrix from which we'll take the most important features.

```
featimp=pd.Series(model.feature_importances_,
                          index=predictor_var).
               sort_values(ascending=False)
print (featimp)
```

Output:

```
Credit_History      0.273094
TotalIncome_log     0.264433
LoanAmount_log      0.229032
Dependents          0.050138
Property_Area       0.048979
Loan_Amount_Term    0.042681
Married             0.025823
Education           0.022426
Gender              0.021895
Self_Employed       0.021500
dtype: float64
```

Let's use the top 5 variables for creating a model. Also, we will modify the parameters of random forest model a little bit:

```
model=RandomForestClassifier(n_estimators=25,
          min_samples_split=25, max_depth=7,
                                max_features=1)
predictor_var=['TotalIncome_log',
     'LoanAmount_log','Credit_History',
                'Dependents','Property_Area']
classification_model(model,df,predictor_var,
                                      outcome_var)
```

Output:

```
Accuracy : 82.899% Cross-Validation Score :
81.461%
```

Notice that although accuracy reduced, but the cross-validation score is improving showing that the model is generalizing well. Remember that random forest models are not exactly repeatable. Different runs will result in slight variations because of randomization. But the output should stay in the ballpark.

You would have noticed that even after some basic parameter tuning on random forest, we have reached a cross-validation accuracy only

slightly better than the original logistic regression model. This exercise gives us some very interesting and unique learning:

1. Using a more sophisticated model does not guarantee better results.
2. Avoid using complex modeling techniques as a black box without understanding the underlying concepts. Doing so would increase the tendency of overfitting thus making your models less interpretable.
3. Feature Engineering is the key to success. Everyone can use an Xgboost models but the real art and creativity lies in enhancing your features to better suit the model.

REFERENCES

1. Eng Michael David. A Practical Introduction to Python Programming. Independently Published, 2020. 264 p.
2. Ben Stephenson. The Python Workbook. A Brief Introduction with Exercises and Solutions. Springer, 2019. 219 p.
3. Andrew Bird, Dr Lau Cher Han, Mario Corchero Jiménez, Graham Lee, Corey Wade. The Python Workshop. A Practical, No-Nonsense Introduction to Python Development. Packt, 2019. 606 p.
4. What is Data Analysis? Research | Types | Methods | Techniques. URL : http://surl.li/nayy.
5. Why Python is Essential for Data Analysis. URL : https://www.rtinsights.com/why-python-is-essential-for-data-analysis.
6. A Complete Python Tutorial to Learn Data Science from Scratch. URL : http://surl.li/nbdy.
7. KDnuggets. URL : https://www.kdnuggets.com/2020/01/decision-tree-algorithm-explained.html.

Навчальне видання

Конспект лекцій
із курсу «Пайтон та наука про дані»

(Англійською мовою)

Відповідальний за випуск І. В. Коплик
Редактор І. А. Іванов
Комп'ютерне верстання І. О. Князя

Формат 60x84/16. Ум. друк. арк. 5,81. Обл.-вид. арк. 5,65.

Видавець і виготовлювач
Сумський державний університет,
вул. Римського-Корсакова, 2, м. Суми, 40007
Свідоцтво суб'єкта видавничої справи ДК № 3062 від 17.12.2007.