Chapter :2

- OBJECT ORIENTED PROGRAMMING

Object-Oriented Programming

- Python is an object-oriented programming language, and we have in fact been using many object-oriented concepts already.
- The key notion is that of an *object*.
- An object consists of two things: data and functions (called *methods*) that work with that data.
- As an example, strings in Python are objects.
- The data of the string object is the actual characters that make up that string. The methods are things like lower, replace, and split.

WHAT ARE ATTRIBUTES?

- data and procedures that "**belong**" to the class
- **data attributes**
  - think of data as other objects that make up the class
  - *for example, a coordinate is made up of two numbers*
- **methods** (procedural attributes)
  - think of methods as functions that only work with this class
  - how to interact with the object
  - *for example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects*

WHAT IS A METHOD?

- procedural attribute, like a **function that works only with this class**
- Python always passes the object as the first argument
  - convention is to use **self** as the name of the first argument of all methods
- the "**.**" **operator** is used to access any attribute
  - a data attribute of an object
  - a method of an object

Object-Oriented Programming cont.…

- In Python, everything is an object. That includes not only strings and lists, but also integers, floats, and even functions themselves.

`

Creating your own classes

- A *class* is a template for objects. It contains the code for all the object's methods.
-

```
class Example:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def add(self):
        return self.a + self.b

e = Example(8, 6)
print(e.add())
```

- To create a class, we use the **class** statement.
- Class names usually start with a capital.
  Most classes will have a method called __init__.
- The underscores indicate that it is a special kind of method. It is called a *constructor*, and it is automatically called when someone creates a new object from your class.
- The constructor is usually used to set up the class's variables.
- In the above program, the constructor takes two values, a and b, and assigns the class variables a and b to those values.

- The first argument to every method in your class is a special variable called self.
- Every time your class refers to one of its variables or methods, it must precede them by self.
- The purpose of self is to distinguish your class's variables and methods from other variables and functions in the program.
- To create a new object from the class, you call the class name along with any values that you want to send to the constructor.
- You will usually want to assign it to a variable name. This is what the line e=Example(8,6) does
- To use the object's methods, use the dot operator, as in e.addmod().

A more practical example

- Here is a class called Analyzer that performs some simple analysis on a string. There are methods to return how many words are in the string, how many are of a given length, and how many start with a given string.

A more practical example cont....

```
from string import punctuation

class Analyzer:
    def __init__(self, s):
        for c in punctuation:
            s = s.replace(c,'')
        s = s.lower()
        self.words = s.split()

    def number_of_words(self):
        return len(self.words)

    def starts_with(self, s):
        return len([w for w in self.words if w[:len(s)]==s])

    def number_with_length(self, n):
        return len([w for w in self.words if len(w)==n])


s = 'This is a test of the class.'
analyzer = Analyzer(s)
print(analyzer.words)
print('Number of words:', analyzer.number_of_words())
print('Number of words starting with "t":', analyzer.starts_with('t'))
print('Number of 2-letter words:', analyzer.number_with_length(2))
```

```
['this', 'is', 'a', 'test', 'of', 'the', 'class']
Number of words: 7
Number of words starting with "t": 3
Number of 2-letter words: 2
```

Inheritance

- In object-oriented programming there is a concept called *inheritance* where you can create a class that builds off of another class .
- When you do this, the new class gets all of the variables and methods of the class it is inheriting from (called the *base class*).
- It can then define additional variables and methods that are not present in the base class, and it can also *override* some of the methods of the base class.
-

Here is a simple example:

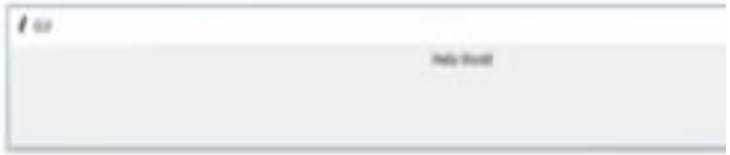
```
class Parent:
    def __init__(self, a):
        self.a = a
    def method1(self):
        print(self.a*2)
    def method2(self):
        print(self.a+'!!!')

class Child(Parent):
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def method1(self):
        print(self.a*7)
    def method3(self):
        print(self.a + self.b)

p = Parent('hi')
c = Child('hi', 'bye')

print('Parent method 1: ', p.method1())
print('Parent method 2: ', p.method2())
print()
print('Child method 1: ', c.method1())
print('Child method 2: ', c.method2())
print('Child method 3: ', c.method3())
```

Chapter 3

Python Graphical User Interface (GUI)

First Window Using Tkinter


```
import tkinter
window = tkinter.Tk()
# to rename the title of the window
window.title("GUI")
# pack is used to show the object in
the window
label = tkinter.Label(window, text =
"Hello World!").pack()
window.mainloop()
```



Adding Widgets To Our Application

A widget is an element of a graphical user interface (GUI) that displays information or provides a specific way for a user to interact with the operating system or an application

GUI Programming with Tkinter

- Up until now, the only way our programs have been able to interact with the user is through keyboard input via the **input** statement.
- But most real programs use windows, buttons, scrollbars, and various other things

GUI Programming with Tkinter cont…

- These *widgets* are part of what is called a *Graphical User Interface* or GUI.
- This chapter is about GUI programming in Python with Tkinter

Basics

- Nearly every GUI program we will write will contain the following three lines:

```
from tkinter import *
root = Tk()

mainloop()
```

Basics cont…

- The first line imports all of the GUI stuff from the tkinter module.
- The second line creates a window on the screen, which we call root.
- The third line puts the program into what is essentially a long-running while loop called the *event loop*. This loop runs, waiting for keypresses, button clicks, etc., and it exits when the user closes the window.

Labels

- A label is a place for your program to place some text on the screen.
- The following code creates a label and places it on the screen.

```
hello_label = Label(text='hello')
hello_label.grid(row=0, column=0)
```

We call Label to create a new label. The capital L is required. Our label's name is hello_label.

Once created, use the grid method to place the label on the screen. We will explain grid in the next section.

Label cont..

- **Options** There are a number of options you can change including font size and color. Here are some examples:

```
hello_label = Label(text='hello', font=('Verdana', 24, 'bold'), bg='blue', fg='white')
```

- ✓ font — The basic structure is font= (*font name, font size, style*). You can leave out the font size or the style.
- ✓ The choices for style are 'bold', 'italic', 'underline', 'overstrike', 'roman', and 'normal' (which is the default). You can combine multiple styles like this: 'bold italic'

Label cont..

- fg and bg — These stand for foreground and background. Many common color names can be used, like 'blue', 'green', etc.
- width — This is how many characters long the label should be. If you leave this out, Tkinter will base the width off of the text you put in the label. This can make for unpredictable results, so it is good to decide ahead of time how long you want your label to be and set the width accordingly.
- height — This is how many rows high the label should be. You can use this for multiline labels. Use newline characters in the text to get it to span multiple lines. For example, text='hi\nthere'.

Changing label properties

- Later in your program, after you've created a label, you may want to change something about it.
- To do that, use its configure method. Here are two examples that change the properties of a label called label:
  label.configure(text='Bye')
  label.configure(bg='white', fg='black')

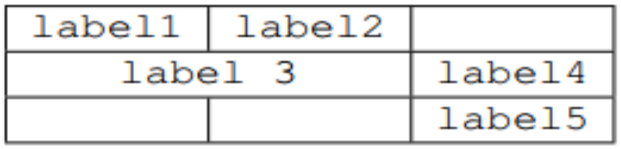
grid

- The grid method is used to place things on the screen. It lays out the screen as a rectangular grid of rows and columns. The first few rows and columns are shown below.

(row=0, column=0)	(row=0, column=1)	(row=0, column=2)
(row=1, column=0)	(row=1, column=1)	(row=1, column=2)
(row=2, column=0)	(row=2, column=1)	(row=2, column=2)

- **Spanning multiple rows or columns** There are optional arguments, rowspan and columnspan, that allow a widget to take up more than one row or column.
- Here is an example of several grid statements followed by what the layout will look like:

```
label1.grid(row=0, column=0)
label2.grid(row=0, column=1)
label3.grid(row=1, column=0, columnspan=2)
label4.grid(row=1, column=2)
label5.grid(row=2, column=2)
```



Entry boxes

- Entry boxes are a way for your GUI to get text input.
- The following example creates a simple entry box and places it on the screen

```
entry = Entry()
entry.grid(row=0, column=0)
```

Entry boxes cont…

- Most of the same options that work with labels work with entry boxes (and most of the other widgets we will talk about).
- The width option is particularly helpful because the entry box will often be wider than you need.

Entry boxes cont....

- **Getting text** To get the text from an entry box, use its `get` method. This will return a string. If you need numerical data, use **`eval`** (or **`int`** or **`float`**) on the string. Here is a simple example that gets text from an entry box named `entry`.

```
string_value = entry.get()
num_value = eval(entry.get())
```

- **Deleting text** To clear an entry box, use the following:

```
entry.delete(0,END)
```

- **Inserting text** To insert text into an entry box, use the following:

```
entry.insert(0, 'hello')
```

Buttons

- The following example creates a simple button:

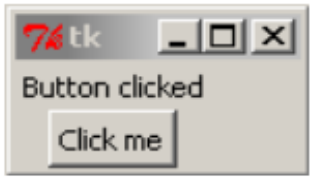
```
ok_button = Button(text='Ok')
```

- To get the button to do something when clicked, use the command argument. It is set to the name of a function, called a *callback function*.
- When the button is clicked, the callback function is called.Here is an example:

Buttons

```
from tkinter import *

def callback():
    label.configure(text='Button clicked')

root = Tk()
label = Label(text='Not clicked')
button = Button(text='Click me', command=callback)

label.grid(row=0, column=0)
button.grid(row=1, column=0)

mainloop()
```

When the program starts, the label says `Click me`. When the button is clicked, the callback function `callback` is called, which changes the label to say `Button clicked`.

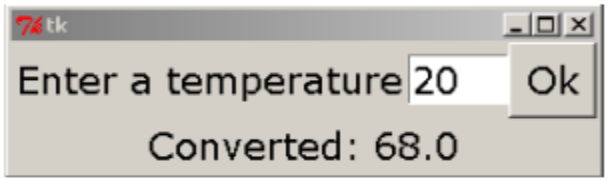


Here is a working GUI program that converts temperatures from Fahrenheit to Celsius.

```
from tkinter import *

def calculate():
    temp = int(entry.get())
    temp = 9/5*temp+32
    output_label.configure(text = 'Converted: {:.1f}'.format(temp))
    entry.delete(0,END)
```

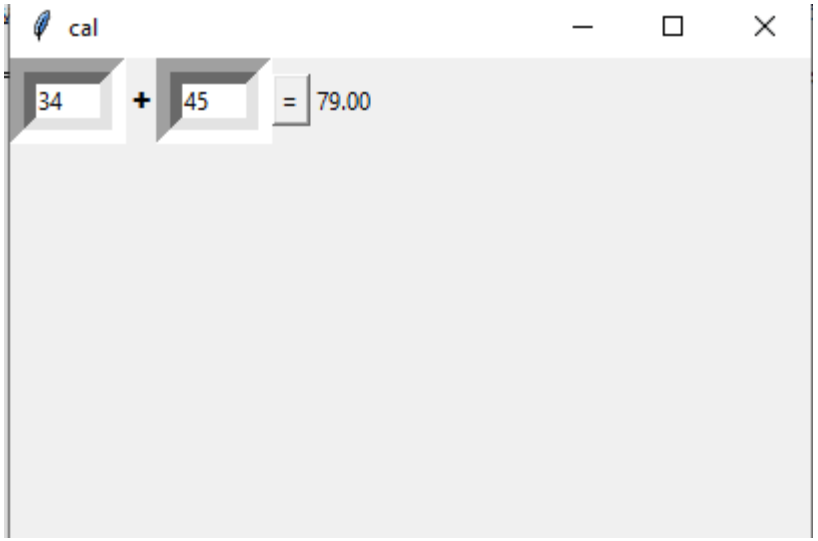
```
root = Tk()
message_label = Label(text='Enter a temperature',
                      font=('Verdana', 16))
output_label = Label(font=('Verdana', 16))
entry = Entry(font=('Verdana', 16), width=4)
calc_button = Button(text='Ok', font=('Verdana', 16),
                     command=calculate)

message_label.grid(row=0, column=0)
entry.grid(row=0, column=1)
calc_button.grid(row=0, column=2)
output_label.grid(row=1, column=0, columnspan=3)

mainloop()
```

Here is what the program looks like:



Quiz 5 marks

- Writes the code to add the number like the figure below?



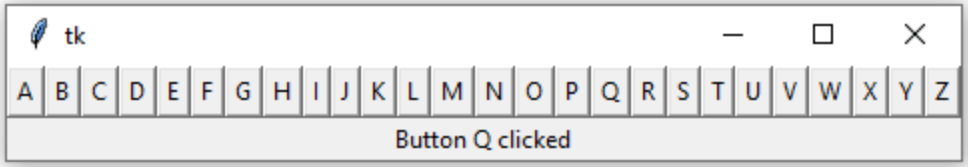
lambda trick

- Sometimes we will want to pass information to the callback function, like if we have several buttons that use the same callback function and we want to give the function information about which button is being clicked.
- Here is an example where we create 26 buttons, one for each letter of the alphabet. Rather than use 26 separate Button() statements and 26 different functions, we use a list and one function.

lambda trick cont..

```
from tkinter import *
alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

def callback(x):
    label.configure(text='Button {} clicked'.format(alphabet[x]))

root = Tk()

label = Label()
label.grid(row=1, column=0, columnspan=26)

buttons = [0]*26 # create a list to hold 26 buttons
for i in range(26):
    buttons[i] = Button(text=alphabet[i],
                        command = lambda x=i: callback(x))
    buttons[i].grid(row=0, column=i)

mainloop()
```

```
========== RESTART: C:/Users/TILAHUN/Desktop/GIS CODE/itkinter.py =
```



Frames

- Let's say we want 26 small buttons across the top of the screen, and a big Ok button below them, like below:

We try the following code:

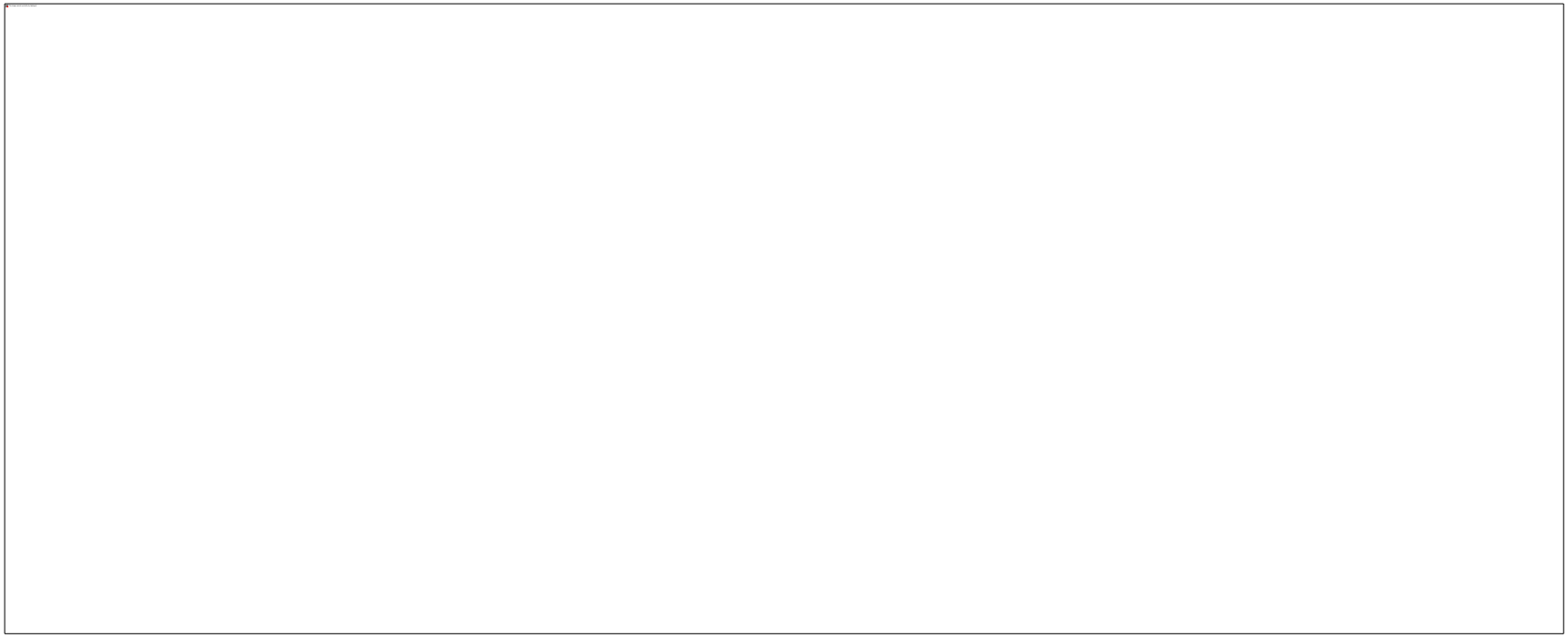
```
from tkinter import *

root = Tk()

alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
buttons = [0]*26
for i in range(26):
    buttons[i] = Button(text=alphabet[i])
    buttons[i].grid(row=0, column=i)

ok_button = Button(text='Ok', font=('Verdana', 24))
ok_button.grid(row=1, column=0)

mainloop()
```

Frames cont…

- But we instead get the following unfortunate result:

- The problem is with column 0. There are two widgets there, the A button and the Ok button, and Tkinter will make that column big enough to handle the larger widget, the Ok button.
- One solution to this problem is shown below: ok_button.grid(row=1, column=0, columnspan=26)

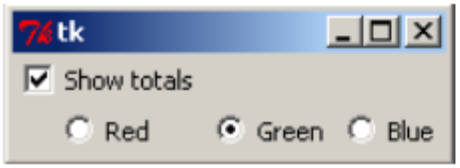
- The frame's job is to hold other widgets and essentially combine them into one large widget.
- In this case, we will create a frame to group all of the letter buttons into one large widget. The code is shown below:



Check buttons and Radio buttons

- The one thing to note here is that we have to tie the check button to a variable, and it can't be just any variable, it has to be a special kind of Tkinter variable, called an IntVar. This variable, show_totals, will be 0 when the check button is unchecked and 1 when it is checked. To access the value of the variable, you need to use its get method, like this:
  show_totals.get()

In the image below, the top line shows a check button and the bottom line shows a radio button.



Check buttons The code for the above check button is:

```
show_totals = IntVar()
check = Checkbutton(text='Show totals', var=show_totals)
```

Check buttons

- You can also set the value of the variable using its set method. This will automatically check or uncheck the check button on the screen. For instance, if you want the above check button checked at the start of the program, do the following:
  show_totals = IntVar()
  show_totals.**set**(1)
  check = Checkbutton(text='Show totals',
  var=show_totals

Radio buttons

- Radio buttons work similarly. The code for the radio buttons shown at the start of the section is:
- color = IntVar()
  redbutton = Radiobutton(text='Red', var=color, value=1)
  greenbutton = Radiobutton(text='Green', var=color, value=2)
  bluebutton = Radiobutton(text='Blue', var=color, value=3)

Radio buttons cont.…

- The value of the IntVar object color will be 1, 2, or 3, depending on whether the left, middle, or
  right button is selected. These values are controlled by the value option, specified when we create
  the radio buttons.

Text widget

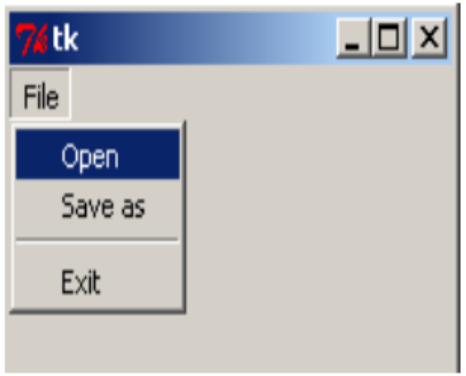
- The Text widget is a bigger, more powerful version of the Entry widget. Here is an example of creating one:
  textbox = Text(font=('Verdana', 16), height=6, width=40)
  The widget will be 40 characters wide and 6 rows tall. You can still type past the sixth row; the widget will just display only six rows at a time, and you can use the arrow keys to scroll.

Text widget cont....

- If you want a scrollbar associated with the text box you can use the ScrolledText widget.
- To use the ScrolledText widget, you will need the following import:
  from tkinter.scrolledtext **import** ScrolledText

Menu bars

We can create a menu bar, like the one below, across the top of a window.



Here is an example that uses some of the dialogs from the previous section:

```
from tkinter import *
from tkinter.filedialog import *

def open_callback():
    filename = askopenfilename()
    # add code here to do something with filename

def saveas_callback():
    filename = asksaveasfilename()
    # add code here to do something with filename

root = Tk()
menu = Menu()
root.config(menu=menu)
file_menu = Menu(menu, tearoff=0)
file_menu.add_command(label='Open', command=open_callback)
file_menu.add_command(label='Save as', command=saveas_callback)
file_menu.add_separator()
file_menu.add_command(label='Exit', command=root.destroy)
menu.add_cascade(label='File', menu=file_menu)

mainloop()
```

Python Regex

- Regex or Regular Expressions are present in every language, be it Java or JavaScript, or any other language. A series of characters defining a search pattern is called a regular expression. These patterns are typically used to "find" or "find and replace" operations on strings or for input validation by string-searching algorithms. It is a methodology developed in formal language theory and theoretical computer science.

Regular Expressions	Description
foo.*	# Matches any string starting with foo
\d*	# Match any number decimal digits
[a-zA-Z]+	# Match a sequence of one or more letters
text	Match literal text
.	Match any character except newline
^	Match the start of a string
$	Match the end of a string
*	Match 0 or more repetitions
+	Match 1 or more repetitions
?	Match 0 or 1 repetition
+?	Match 1 or more, as few as possible
*?	Match 0 or more, as few as possible
{m,n}	Match m to n repetitions
{m,n}?	Match m to n repetitions, few as possible
[...]	Match a set of characters
[^...]	Match characters, not in a set

A | B	Match A or B (...) Match regex in parenthesis as a group
\number	Matches text matched by the previous group
\A	Matches start of the string
\b	Matches empty string at beginning or end of the word
\B	Matches empty string not at begin or end of the word
\d	Matches any decimal digit
\D	Matches any non-digit
\s	Matches any whitespace
\S	Matches any non-whitespace
\w	Matches any alphanumeric character
\W	Matches characters not in
\w \Z	Match at end of the string.
\\	Literal backslash

Form validation

```
from tkinter import *
import re
from tkinter import messagebox
def fun():
    s='^[a-z0-9]+[\._]?[a-z0-9]+[@]\w+[.]\w{2,3}$'

    v1=enty1.get()
    v2=enty2.get()
    v3=enty3.get()
    if(v1==""):
        messagebox.showinfo("showinfo","user name is not fill")
    elif(re.search(s,v2)is None):
        messagebox.showinfo("showinfo","email address is not the correct format")
    elif( v3==""):
        messagebox.showinfo("showinfo","password must be filled")
    elif( len (v3)<=8):
        messagebox.showinfo("showinfo","password must >8 char")
    else:
        messagebox.showinfo("showinfo","login is success ful")

root=Tk()
root.geometry("400x300")
root.title("Form validation")
lab1=Label(text="user name",font=("timesnewroman",16,"bold"))
lab2=Label(text="Email address",font=("timesnewroman",16,"bold"))
lab3=Label(text="password",font=("timesnewroman",16,"bold"))
enty1=Entry(width=14)
enty2=Entry(width=14)
enty3=Entry(width=14)
btn=Button(text="OK",command= fun)
lab1.grid(row=0,column=0)
enty1.grid(row=0,column=1)
lab2.grid(row=1,column=0)
enty2.grid(row=1,column=1)
lab3.grid(row=2,column=0)
enty3.grid(row=2,column=1)
enty3.configure(show="*")
btn.grid(row=3,column=1)

mainloop
```

Chapter 4: Using Database in Python

- What Is DBMS?
- An example of DBMS program ?

- A DBMS is a set of programs that is used to store and manipulation data.

❖Manipulation of data include the following:

✓ Adding new data, for example adding details of new student.

✓ Deleting unwanted data, for example deleting the details of students who have completed course.

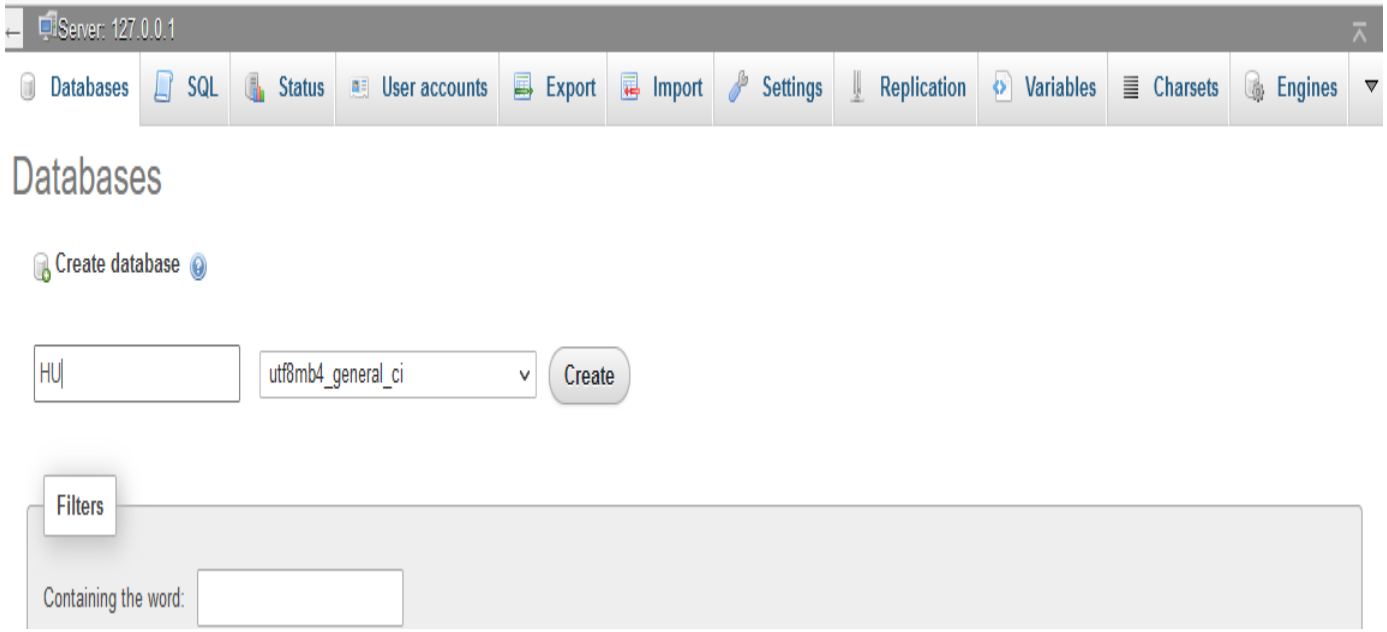
✓ Changing existing data, for example modifying the fee paid by the student.

Extract Data from Database using MySQL-Connector and XAMPP in Python

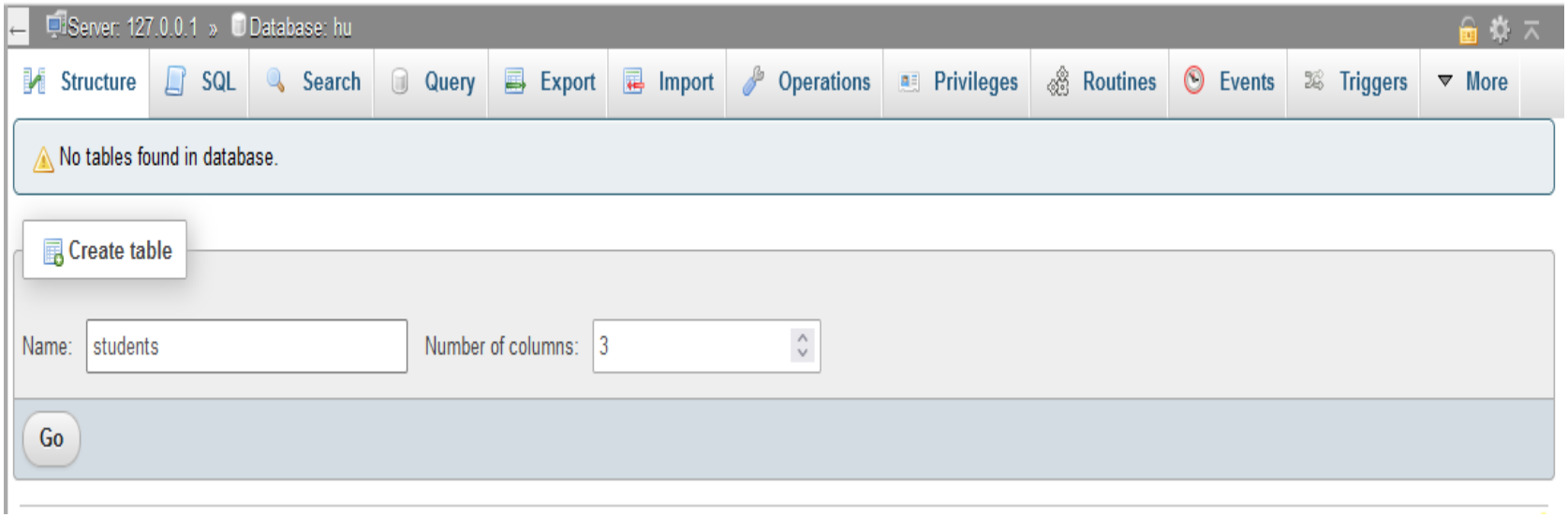
- **Prerequisites:** MySQL-Connector, XAMPP Installation
- A connector is employed when we have to use MySQL with other programming languages. The work of mysql-connector is to provide access to MySQL Driver to the required language. Thus, it generates a connection between the programming language and the MySQL Server.
- **Requirements**
- **XAMPP:** Database / Server to store and display data.
- **MySQL-Connector module:** For connecting the database with the python file. Use the below command to install this module.
- pip install mysql-connector-python

Step-by-step Approach:

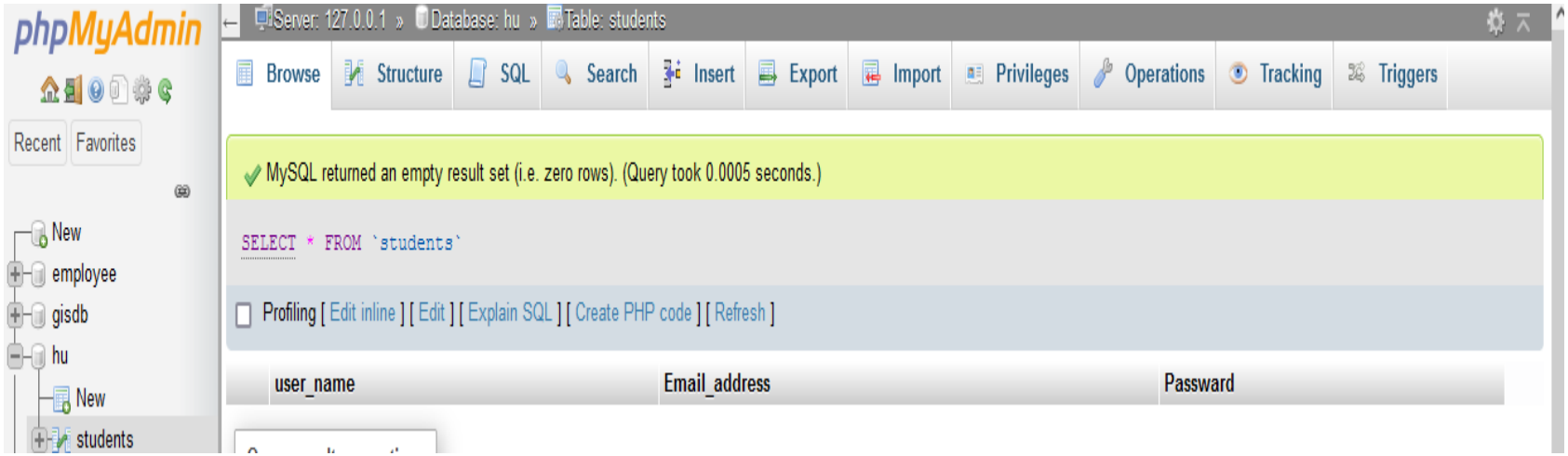
- **Procedure to create a table in the database:**
- Start your *XAMPP* web server.
- Type http://localhost/phpmyadmin/ in your browser.
- Go to Database create database with name and click on Create.



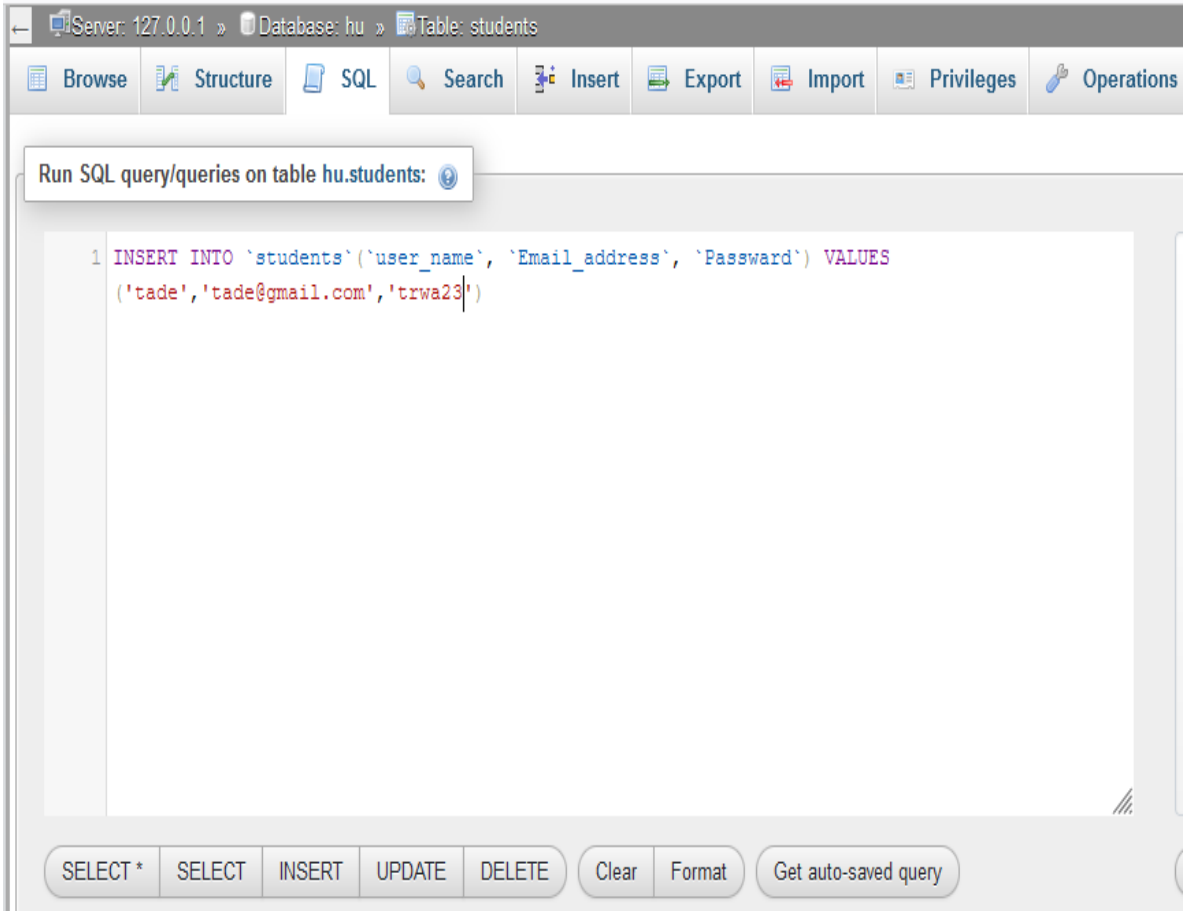
- Create a table with in *HU* database and click on Go.



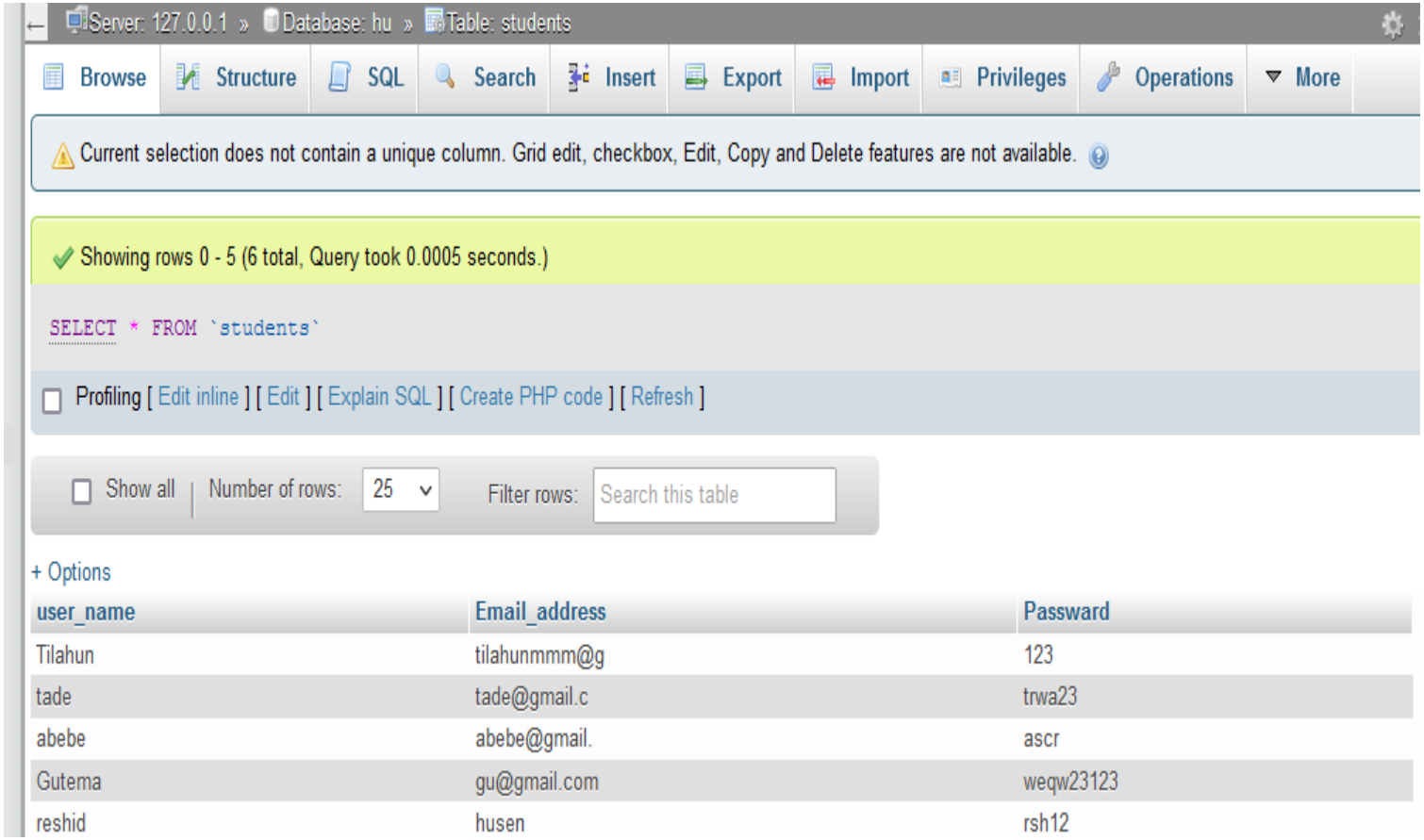
- Your table is created.



Insert data in your database by clicking on *SQL* tab then select *INSERT*



The data in your table is:



Now you can perform operation IE display data in your web page using python

- **Procedure for writing Python program:**
- Import *mysql* connector module in your Python code.
- import mysql.connector
- Create connection object.
- conn_object=mysql.connector.connect(hostname,username,password,database_name)
- Here, you will need to pass server name, username, password, and database name)
- Create a cursor object.
- cur_object=conn_object.cursor()
- Perform queries on database.
- query=DDL/DML etc
- cur_obj=execute(query)
- Close cursor object.
- cur_obj.close()
- Close connection object.
- conn_obj.close()

- code1

Chapter 5: Using Important Python Packages

- Python Packages
- NumPy
- Pandas
- Python Image Library

NumPy Basics: Arrays and Vectorized Computation

- NumPy, short for Numerical Python, is one of the most important foundational packages for numerical computing in Python.
- Most computational packages providing scientific functionality use NumPy's array objects as the *lingua franca(A common language)* for data exchange.

Here are some of the things you'll find in NumPy:

- ✓ ndarray, an efficient multidimensional array providing fast array-oriented arithmetic operations and flexible *broadcasting* capabilities.
- ✓ Mathematical functions for fast operations on entire arrays of data without having to write loops.
- ✓ Tools for reading/writing array data to disk and working with memory mapped files.
- ✓ Linear algebra, random number generation, and Fourier transform capabilities.

Array-oriented computing in Python

- One of the reasons NumPy is so important for numerical computations in Python is because it is designed for efficiency on large arrays of data.
- There are a number of reasons for this:

✓ NumPy internally stores data in a contiguous block of memory, independent of other built-in Python objects.

✓ NumPy's library of algorithms written in the C lan- guage can operate on this memory without any type checking or other overhead. NumPy arrays also use much less memory than built-in Python sequences.

✓ NumPy operations perform complex computations on entire arrays without the need for Python for loops

NumPy provides a number of ways to create arrays in addition to the normal array() function.

- The four most common convenience functions are
- arange();
- zeros();
- ones(), and empty().
- The arange() function takes a start, stop, and step and works exactly like Python's range() function, except that it returns an ndarray.
- The zeros() and ones() functions take an integer or tuple of integers and return an ndar ray whose shape matches that of the tuple and whose elements are all either zero or one.
- The empty() function, on the other hand, will simply allocate memory without assigning it any values. This means that the contents of an empty array will be what-ever happened to be in memory at the time.
- Empty arrays are therefore most useful if you have existing data you want to load into an array, and you do not want to pay the cost of setting all the values to zero if you are just going to overwrite them.
- Here are some examples of how to create new arrays using the arange(), zeros(), ones(), and empty() functions:

NumPy code

- **import numpy as np**
  np.arange(6)
  np.zeros(4)
  np.ones((2, 3))
  np.empty(4)

Important ndarray attributes

Attribute	Description
`data`	Buffer to the raw array data
`dtype`	Type information about what is in `data`
`base`	Pointer to another array where data is stored, or None if data is stored here
`ndim`	Number of dimensions (`int`)
`shape`	Tuple of integers that represents the rank along each dimension; has length of `ndim`
`size`	Total number of elements (`int`), equal to the product of all of the elements of `shape`
`itemsize`	Number of bytes per element (`int`)

Modifying the attributes

- Modifying the attributes in an allowable way will automatically update the values of the other attributes. Since the data buffer is fixed-length, all modifications must pre- serve the size of the array.
- This fixed size restriction also implies that you cannot append to an existing array without copying memory.
- A common method of reshaping an existing array is to assign a new tuple of integers to the shape attribute.
- This will change the shape in-place. For example:
  a = np.arange(4)
  a.shape = (2, 2)

dtypes

- The dtype or data type is the most important ndarray attribute.
- The data type deter-mines the size and meaning of each element of the array.
- The default system of dtypes that NumPy provides is more precise and broader for basic types than the type system that the Python language implements.

Basic NumPy dtypes

dtype	Code	Bytes	Python	Description
`bool_`	?	1	`bool`	Boolean data type. Note that this takes up a full byte (8 bits) and is somewhat inefficient at storing a large number of `bool`s. For a memory-efficient Boolean array, please see Ilan Schnell's bitarray package.
`bool8`	?	1	`bool`	Alias to `bool_`.
`int_`			`int`	Default integer type; alias to either `int32` or `int64`.
`int0`			`int`	Same as `int_`.
`int8`	b	1	`int`	Single-byte (8-bit) integer ranging from `-128` to `127`. Interchangeable with the C/C++ `char` type.

Basic NumPy dtypes cont…

dtype	Code	Bytes	Python	Description
`float_`	`d`	8	`float`	Alias to `float64`.
`float16`	`e`	2	`float`	16-bit floating-point number.
`float32`	`f`	4	`float`	32-bit floating-point number. Usually compatible with the C/C++ `float` type.
`float64`	`d`	8	`float`	64-bit floating-point number. Usually compatible with the C/C++ dou ble type.

Basic NumPy dtypes cont…

- When you are creating an array, the dtype that is automatically selected will always be that of the least precise element.
- Say you have a list that is entirely integers with the exception of a single float. An array created from this list will have the dtype np.float64, because floats are less precise than integers.
- a = np.array([6, 28, 496, 8128])

  a.dtype

  b = np.array([6, 28.0, 496, 8128])

  b.dtype

Slicing and Views

- NumPy arrays have the same slicing semantics as Python lists when it comes to accessing elements or sub arrays.
- 1D Array:

  start: inclusive, starting index

  End : exclusive , ending index

  step : difference between index

Slicing and Views

- NumPy arrays have the same slicing semantics as Python lists when it comes to accessing elements or sub arrays.
-

Code	Returns
`a = np.arange(8)`	`array([0, 1, 2, 3, 4, 5, 6, 7])`
`a[::-1]`	`array([7, 6, 5, 4, 3, 2, 1, 0])`
`a[2:6]`	`array([2, 3, 4, 5])`
`a[1::3]`	`array([1, 4, 7])`

Slicing and Views cont....

Code	Returns

`a = np.arange(16)`
`a.shape = (4, 4)` ❶
	`array([[ 0,  1,  2,  3],`
	`[ 4,  5,  6,  7],`
	`[ 8,  9, 10, 11],`
	`[12, 13, 14, 15]])`

`a[::2, 1::2]` ❷	`array([[ 1,  3],` `[ 9, 11]])`
`a[1:3, 1:3]` ❸	`array([[ 5,  6],` `[ 9, 10]])`

`a[2::-1, :3]` ❹	`array([[ 8,  9, 10],`
	`[ 4,  5,  6],`
	`[ 0,  1,  2]])`

❶ Create a 1D array and reshape it to be 4x4.

❷ Slice the even rows and the odd columns.

❸ Slice the inner 2x2 array.

❹ Reverse the first 3 rows, taking the first 3 columns.

Slicing and Views cont.…

- The most important feature of array slicing to understand is that slices are *views* into the original array. No data is copied when a slice is made, making NumPy especially fast for slicing operations

Arithmetic and Broadcasting

- A defining feature of all array data languages is the ability to perform arithmetic operations in an *element-wise* fashion.
- This allows for concise mathematical expressions to be evaluated over an arbitrarily large amount of data.
- This works equally well for scalars as it does for arrays with the same shape.

Arithmetic and Broadcasting cont..

- In the following example, we see how simple arithmetic operations (addition, subtraction, multiplication, etc.) are evaluated with an array as a variable:
- a = np.arange(6)
  a - 1
  a + a
  2*a**2 + 3*a + 1

repeating Elements: tile and repeat

- Two useful tools for repeating or replicating arrays to produce larger arrays are the repeat and tile functions. repeat replicates each element in an array some number of times, producing a larger array:

```
arr = np.arange(3)

arr
array([0, 1, 2])

arr.repeat(3)
array([0, 0, 0, 1, 1, 1, 2, 2, 2])
```

```
arr.repeat([2, 3, 4])
array([0, 0, 1, 1, 1, 2, 2, 2, 2])
```

pandas

- It contains data structures and data manipulation tools designed to make data cleaning and analysis fast and easy in Python. pandas is often used in tandem with numerical computing tools like NumPy .
- While pandas adopts many coding idioms from NumPy, the biggest difference is that pandas is designed for working with tabular or heterogeneous data.
- NumPy, by contrast, is best suited for working with homogeneous numerical array data
- use the following import convention for pandas:

convention for pandas

- **import pandas as pd**
  Thus, whenever you see pd. in code, it's referring to pandas.
- You may also find it easier to import Series and DataFrame into the local namespace since they are so fre-quently used:

- **from pandas import** Series, DataFrame

pandas Data Structures

- To get started with pandas, you will need to get comfortable with its two workhorse data structures: *Series* and *DataFrame*.
- Series

✓ A Series is a one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) and an associated array of data labels, called its *index*.

✓ The simplest Series is formed from only an array of data:

```
In [11]: obj = pd.Series([4, 7, -5, 3])

In [12]: obj
Out[12]:
0    4
1    7
2   -5
3    3
dtype: int64
```

pandas Data Structures

- The string representation of a Series displayed interactively shows the index on the left and the values on the right. Since we did not specify an index for the data, default one consisting of the integers 0 through N - 1 (where N is the length of the data) is created.
- You can get the array representation and index object of the Series via its values and index attributes, respectively:

```
In [13]: obj.values
Out[13]: array([ 4,  7, -5,  3])

In [14]: obj.index  # like range(4)
Out[14]: RangeIndex(start=0, stop=4, step=1)
```

pandas Data Structures cont....

- Often it will be desirable to create a Series with an index identifying each data point with a label:

```
In [15]: obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

In [16]: obj2
Out[16]:
d    4
b    7
a   -5
c    3
dtype: int64

In [17]: obj2.index
Out[17]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

pandas Data Structures cont....

- Compared with NumPy arrays, you can use labels in the index when selecting single values or a set of values:

```
In [18]: obj2['a']
Out[18]: -5

In [19]: obj2['d'] = 6

In [20]: obj2[['c', 'a', 'd']]
Out[20]:
c    3
a   -5
d    6
dtype: int64
```

Here ['c', 'a', 'd'] is interpreted as a list of indices, even though it contains strings instead of integers.

pandas Data Structures cont.…

- Using NumPy functions or NumPy-like operations, such as filtering with a boolean array, scalar multiplication, or applying math functions, will preserve the index-value link:

```
In [21]: obj2[obj2 > 0]
Out[21]:
d    6
b    7
c    3
dtype: int64

In [22]: obj2 * 2
Out[22]:
d    12
b    14
a   -10
c     6
dtype: int64

In [23]: np.exp(obj2)
Out[23]:
d      403.428793
b     1096.633158
a        0.006738
c       20.085537
dtype: float64
```

pandas Data Structures cont....

- Another way to think about a Series is as a fixed-length, ordered dict, as it is a map- ping of index values to data values. It can be used in many contexts where you might use a dict:

```
In [24]: 'b' in obj2
Out[24]: True
```

```
In [25]: 'e' in obj2
Out[25]: False
```

pandas Data Structures cont....

- Should you have data contained in a Python dict, you can create a Series from it by passing the dict:

```
In [26]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

In [27]: obj3 = pd.Series(sdata)

In [28]: obj3
Out[28]:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

pandas Data Structures cont....

- When you are only passing a dict, the index in the resulting Series will have the dict's keys in sorted order. You can override this by passing the dict keys in the order you want them to appear in the resulting Series:

```
In [29]: states = ['California', 'Ohio', 'Oregon', 'Texas']

In [30]: obj4 = pd.Series(sdata, index=states)

In [31]: obj4
Out[31]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

pandas Data Structures cont....

- Here, three values found in sdata were placed in the appropriate locations, but since no value for 'California' was found, it appears as NaN (not a number), which is con- sidered in pandas to mark missing or *NA* values. Since 'Utah' was not included in states, it is excluded from the resulting object.
- I will use the terms “missing” or “NA” interchangeably to refer to missing data.
- The isnull and notnull functions in pandas should be used to detect missing data:

```
In [32]: pd.isnull(obj4)
Out[32]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool

In [33]: pd.notnull(obj4)
Out[33]:

California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

Series also has these as instance methods:

```
In [34]: obj4.isnull()
Out[34]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

Data Frame

- A DataFrame represents a rectangular table of data and contains an ordered collec-tion of columns, each of which can be a different value type (numeric, string, Boolean, etc.).
- The DataFrame has both a row and column index; it can be thought of
as a dict of Series all sharing the same index.
- There are many ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays:
-

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

Data Frame cont..

The resulting DataFrame will have its index assigned automatically as with Series, and the columns are placed in sorted order:

```
In [45]: frame
Out[45]:
   pop   state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  3.6    Ohio  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002
5  3.2  Nevada  2003
```

For large DataFrames, the head method selects only the first five rows:

```
In [46]: frame.head()
Out[46]:
   pop   state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  3.6    Ohio  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002
```

Data Frame cont..

If you specify a sequence of columns, the DataFrame's columns will be arranged in that order:

```
In [47]: pd.DataFrame(data, columns=['year', 'state', 'pop'])
Out[47]:
   year   state  pop
0  2000    Ohio  1.5
1  2001    Ohio  1.7
2  2002    Ohio  3.6
3  2001  Nevada  2.4
4  2002  Nevada  2.9
5  2003  Nevada  3.2
```

If you pass a column that isn't contained in the dict, it will appear with missing values in the result:

```
In [48]: frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
   ....:                        index=['one', 'two', 'three', 'four',
   ....:                               'five', 'six'])
```

Data Frame cont..

```
In [49]: frame2
Out[49]:
       year   state  pop debt
one    2000    Ohio  1.5  NaN
two    2001    Ohio  1.7  NaN
three  2002    Ohio  3.6  NaN
four   2001  Nevada  2.4  NaN
five   2002  Nevada  2.9  NaN
six    2003  Nevada  3.2  NaN

In [50]: frame2.columns
Out[50]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

column in a DataFrame can be retrieved as a Series either by dict-like notation or attribute:

```
In [51]: frame2['state']
Out[51]:
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object
```

Data Frame cont..

Columns can be modified by assignment. For example, the empty 'debt' column could be assigned a scalar value or an array of values:

```
In [54]: frame2['debt'] = 16.5

In [55]: frame2
Out[55]:
       year   state  pop  debt
one    2000    Ohio  1.5  16.5
two    2001    Ohio  1.7  16.5
three  2002    Ohio  3.6  16.5
four   2001  Nevada  2.4  16.5
five   2002  Nevada  2.9  16.5
six    2003  Nevada  3.2  16.5

In [56]: frame2['debt'] = np.arange(6.)

In [57]: frame2
Out[57]:
       year   state  pop  debt
one    2000    Ohio  1.5   0.0
two    2001    Ohio  1.7   1.0
three  2002    Ohio  3.6   2.0
four   2001  Nevada  2.4   3.0
five   2002  Nevada  2.9   4.0
six    2003  Nevada  3.2   5.0
```

Data Frame cont..

- When you are assigning lists or arrays to a column, the value's length must match the length of the DataFrame. If you assign a Series, its labels will be realigned exactly to the DataFrame's index, inserting missing values in any holes:

```
In [58]: val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])

In [59]: frame2['debt'] = val

In [60]: frame2
Out[60]:
       year   state  pop  debt
one    2000    Ohio  1.5   NaN
two    2001    Ohio  1.7  -1.2
three  2002    Ohio  3.6   NaN
four   2001  Nevada  2.4  -1.5
five   2002  Nevada  2.9  -1.7
six    2003  Nevada  3.2   NaN
```

Data Frame cont..

- Assigning a column that doesn’t exist will create a new column. The del keyword will delete columns as with a dict. As an example of del, I first add a new column of boolean values where the state column equals 'Ohio':

```
In [61]: frame2['eastern'] = frame2.state == 'Ohio'

In [62]: frame2
Out[62]:
       year   state  pop  debt  eastern
one    2000    Ohio  1.5   NaN     True
two    2001    Ohio  1.7  -1.2     True
three  2002    Ohio  3.6   NaN     True
four   2001  Nevada  2.4  -1.5    False
five   2002  Nevada  2.9  -1.7    False
six    2003  Nevada  3.2   NaN    False
```

Data Frame cont..

The del method can then be used to remove this column:

```
In [63]: del frame2['eastern']

In [64]: frame2.columns
Out[64]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

Essential Functionality

- Reindexing
  An important method on pandas objects is reindex, which means to create a new object with the data *conformed* to a new index. Consider an example:
- In [91]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
  In [92]: obj

Essential Functionality cont..

Calling `reindex` on this Series rearranges the data according to the new index, introducing missing values if any index values were not already present:

```
In [93]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

In [94]: obj2
Out[94]:
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

Essential Functionality cont..

With DataFrame, `reindex` can alter either the (row) index, columns, or both. When passed only a sequence, it reindexes the rows in the result:

```
In [98]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
   ....:                      index=['a', 'c', 'd'],
   ....:                      columns=['Ohio', 'Texas', 'California'])

In [99]: frame
Out[99]:
   Ohio  Texas  California
a     0      1           2
c     3      4           5
d     6      7           8
```

Indexing, Selection, and Filtering

- Series indexing (obj[...]) works analogously to NumPy array indexing, except you can use the Series's index values instead of only integers. Here are some examples of this:

```
In [117]: obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])

In [118]: obj
Out[118]:
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64
```

```
In [119]: obj['b']
Out[119]: 1.0
```

```
In [121]: obj[2:4]
Out[121]:
c    2.0
d    3.0
dtype: float64
```

Indexing, Selection, and Filtering

- Slicing with labels behaves differently than normal Python slicing in that the end-point is inclusive:

```
In [125]: obj['b':'c']
Out[125]:
b    1.0
c    2.0
dtype: float64
```

Indexing, Selection, and Filtering cont..

Setting using these methods modifies the corresponding section of the Series:

```
In [126]: obj['b':'c'] = 5

In [127]: obj
Out[127]:
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

Arithmetic and Data Alignment

- An important pandas feature for some applications is the behavior of arithmetic between objects with different indexes.
-

```
In [150]: s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])

In [151]: s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
   .....:                index=['a', 'c', 'e', 'f', 'g'])

In [152]: s1
Out[152]:
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64

In [153]: s2
Out[153]:
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

Adding these together yields:

```
In [154]: s1 + s2
Out[154]:
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

In the case of DataFrame, alignment is performed on both the rows and the columns:

```
In [155]: df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
   .....:                    index=['Ohio', 'Texas', 'Colorado'])

In [156]: df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
   .....:                    index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [157]: df1
Out[157]:
            b    c    d
Ohio      0.0  1.0  2.0
Texas     3.0  4.0  5.0
Colorado  6.0  7.0  8.0

In [158]: df2
Out[158]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0
```

Adding these together returns a DataFrame whose index and columns are the unions of the ones in each DataFrame:

```
In [159]: df1 + df2
Out[159]:
            b   c     d   e
Colorado  NaN NaN   NaN NaN
Ohio      3.0 NaN   6.0 NaN
Oregon    NaN NaN   NaN NaN
Texas     9.0 NaN  12.0 NaN
Utah      NaN NaN   NaN NaN
```

Since the `'c'` and `'e'` columns are not found in both DataFrame objects, they appear as all missing in the result. The same holds for the rows whose labels are not common to both objects.

Sorting

Sorting a dataset by some criterion is another important built-in operation. To sort lexicographically by row or column index, use the `sort_index` method, which returns a new, sorted object:

```
In [201]: obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])

In [202]: obj.sort_index()
Out[202]:
a    1
b    2
c    3
d    0
dtype: int64
```