Module 1: Python Programming OOP, GUI and Database Connectivity

Training Lab Setup

1. Install Python and set global
2. Install XAMPP
3. Install VSCode
4. Download Datasets

Functions, Classes and Objects

Native Functions

```
# using datetime
dateref=datetime.datetime.now()
print('year:' + dateref.strftime('%Y'))
print('month:' + dateref.strftime('%m'))
print('month name short:' + dateref.strftime('%b'))
print('month name long:' + dateref.strftime('%B'))
print('day:' + dateref.strftime('%Y'))
print('day of week:' + dateref.strftime('%A'))
print('hour (24 hour):' + dateref.strftime('%H'))
print('hour (12 hour):' + dateref.strftime('%I'))
print('minutes:' + dateref.strftime('%M'))
print('seconds:' + dateref.strftime('%S'))
print('time of day:' + dateref.strftime('%I:%M %p')) #%p must be used with 12-hour format %I
```

```
# date difference
import datetime

date1=datetime.datetime.strptime('October 13, 1985 8:00AM','%B %d, %Y %I:%M%p')
date2=datetime.datetime.now()

diff=date2-date1
print("difference in years:" + str(int(diff.days/365)) + "\n")
print("difference in hours:" + str(int(diff.total_seconds()/3600)) + "\n")
```

Mathematical Functions

Function	**Returns (Description)**
abs(x)	The absolute value of x: the (positive) distance between x and zero.
ceil(x)	The ceiling of x: the smallest integer not less than x.
exp(x)	The exponential of x: ex
fabs(x)	The absolute value of x.
floor(x)	The floor of x: the largest integer not greater than x.
log(x)	The natural logarithm of x, for x> 0.
log10(x)	The base-10 logarithm of x for x> 0.
max(x1, x2,...)	The largest of its arguments: the value closest to positive infinity.
min(x1, x2,...)	The smallest of its arguments: the value closest to negative infinity.
modf(x)	The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.
pow(x, y)	The value of x**y.
round(x [,n])	x rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: round(0.5) is 1.0 and round(-0.5) is -1.0.
sqrt(x)	The square root of x for x > 0.

```
# Random Number Functions
```

Function	Description
choice(seq)	A random item from a list, tuple, or string.
randrange ([start,] stop [,step])	A randomly selected element from range(start, stop, step).
random()	A random float r, such that 0 is less than or equal to r and r is less than 1.
seed([x])	Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None.
shuffle(lst)	Randomizes the items of a list in place. Returns None.
uniform(x, y)	A random float r, such that x is less than or equal to r and r is less than y.

Accessing Values in Strings

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring.

```
var1 = 'Hello World!'
var2 = "Python Programming"
print ("var1[0]: ", var1[0])
print ("var2[1:5]: ", var2[1:5])
```

Updating Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether.

```
var1 = 'Hello World!'
print ("Updated String :- ", var1[:6] + 'Python')
```

Escape Characters

Following is a list of escape or non-printable characters that can be represented with backslash notation.
An escape character gets interpreted; in a single quoted as well as double quoted strings.

Backslash notation	Hexadecimal character	Description
a	0x07	Bell or alert
b	0x08	Backspace
\cx		Control-x
\C-x		Control-x
\e	0x1b	Escape
\f	0x0c	Form feed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0.7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

Using variables in strings

```
print ("My name is %s and weight is %d kg!" % ('Zara', 21))
print (f"My name is {var1} and weight is {var2} kg!")
```

Raw strings do not treat the backslash as a special character at all. Every character you put into a raw string stays the way you wrote it-

```
print ('C:\\nowhere')
```

When the above code is executed, it produces the following result-

```
C:\nowhere
```

Now let us make use of raw string. We would put expression in r'expression' as follows-

```
print (r'C:\\nowhere')
```

When the above code is executed, it produces the following result-

```
C:\\nowhere
```

Built-in String Methods

Methods	Description
capitalize()	Capitalizes first letter of string
center(width, fillchar)	Returns a string padded with fillchar with the original string centered to a total of width columns.
count(str, beg= 0,end=len(string))	Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
decode(encoding='UTF-8',errors='strict')	Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
encode(encoding='UTF-8',errors='strict')	Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.
endswith(suffix, beg=0, end=len(string))	Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
expandtabs(tabsize=8)	Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
find(str, beg=0 end=len(string))	Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
index(str, beg=0, end=len(string))	Same as find(), but raises an exception if str not found.
isalnum()	Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
isalpha()	Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
isdigit()	Returns true if the string contains only digits and false otherwise.
islower()	Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
isnumeric()	Returns true if a unicode string contains only numeric characters and false otherwise.
isspace()	Returns true if string contains only whitespace characters and false otherwise.
istitle()	Returns true if string is properly "titlecased" and false otherwise.
isupper()	Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
join(seq)	Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
len(string)	Returns the length of the string
ljust(width[, fillchar])	Returns a space-padded string with the original string left-justified to a total of width columns.
lower()	Converts all uppercase letters in string to lowercase.
lstrip()	Removes all leading whitespace in string.
maketrans()	Returns a translation table to be used in translate function.
max(str)	Returns the max alphabetical character from the string str.
min(str)	Returns the min alphabetical character from the string str.
replace(old, new [, max])	Replaces all occurrences of old in string with new or at most max occurrences if max given.
rfind(str, beg=0,end=len(string))	Same as find(), but search backwards in string.
rindex(str, beg=0, end=len(string))	Same as index(), but search backwards in string.
rjust(width,[, fillchar])	Returns a space-padded string with the original string right-justified to a total of width columns.
rstrip()	Removes all trailing whitespace of string.

split(str="", num=string.count(str))	Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
splitlines(num=string.count('\n'))	Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.
startswith(str, beg=0,end=len(string))	Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.
strip([chars])	Performs both lstrip() and rstrip() on string
swapcase()	Inverts case for all letters in string.
title()	Returns "title cased" version of string, that is, all words begin with uppercase and the rest are lowercase.
translate(table, deletechars="")	Translates string according to translation table str(256 chars), removing those in the del string.
upper()	Converts lowercase letters in string to uppercase.
zfill (width)	Returns original string left padded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).
isdecimal()	Returns true if a unicode string contains only decimal characters and false otherwise.

User-Defined Functions

The following function takes a string as input parameter and prints it on the standard screen.

```
def printme( str ):
  print (str)
  return
```

Calling a Function

```
def printme( str ):
  print (str)
  return

printme("This is first call to the user defined function!")
printme("Again second call to the same function")
```

Pass by Reference vs Value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example-

```
def changeme( mylist ):
  print ("Values inside the function before change: ", mylist)
  mylist[2]=50
  print ("Values inside the function after change: ", mylist)
  return

# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

```
def changeme( mylist ):
  mylist = [1,2,3,4] # This would assi new reference in mylist
  print ("Values inside the function: ", mylist)
  return
```

```
# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

The parameter mylist is local to the function changeme. Changing mylist within the function does not affect mylist. The function accomplishes nothing and finally this would produce the following result-

```
Values inside the function: [1, 2, 3, 4]
Values outside the function: [10, 20, 30]
```

Required Arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function printme(), you definitely need to pass one argument, otherwise it gives a syntax error as follows-

```
def printme( str ):
  print (str)
  return

# Now you can call printme function
printme()
```

When the above code is executed, it produces the following result-

```
Traceback (most recent call last):
File "test.py", line 11, in <module>
printme()
TypeError: printme() missing 1 required positional argument: 'str'
```

Keyword Arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name. This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the printme() function in the following ways-

```
def printme( str ):
  print (str)
  return

# Now you can call printme function
printme( str = "My string")
```

The following example gives a clearer picture. Note that the order of parameters does not matter.

```
def printinfo( name, age ):
  print ("Name: ", name)
  print ("Age ", age)
  return

# Now you can call printinfo function
printinfo( age=50, name="miki" )
```

Default Arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed.

```
def printinfo( name, age = 35 ):
  print ("Name: ", name)
  print ("Age ", age)
```

```
  return

# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

Variable-length Arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.

An asterisk (*) is placed before the variable name that holds the values of all non-keyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

```
def printinfo( arg1, *vartuple ):
  print ("Output is: ")
  print (arg1)
  for var in vartuple:
    print (var)
  return

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

Classes and Objects

Creating Classes

Following is an example of a simple Python class

```
class Employee:
  'Common base class for all employees'
  empCount = 0

  def __init__(self, name, salary):
    self.name = name
    self.salary = salary

    Employee.empCount += 1

  def displayCount(self):
    print "Total Employee %d" % Employee.empCount

  def displayEmployee(self):
    print ("Name : ", self.name, ", Salary: ", self.salary)
```

The variable empCount is a class variable whose value is shared among all the instances of a in this class. This can be accessed as Employee.empCount from inside the class or outside the class.

- The first method __init__() is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is self. Python adds the self argument to the list for you; you do not need to include it when you call the methods.

Creating Instance Objects

To create instances of a class, you call the class using class name and pass in whatever arguments its __init__ method accepts.

This would create first object of Employee class

```
emp1 = Employee("Zara", 2000)
```

This would create second object of Employee class

```
emp2 = Employee("Manni", 5000)
```

Accessing Attributes

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows

```
emp1.displayEmployee()
emp2.displayEmployee()
print ("Total Employee %d" % Employee.empCount)
```

You can add, remove, or modify attributes of classes and objects at any time

```
emp1.salary = 7000       # Add an 'salary' attribute.
emp1.name = 'xyz'        # Modify 'age' attribute.
del emp1.salary # Delete 'age' attribute.
```

Instead of using the normal statements to access attributes, you can use the following functions-

- The getattr(obj, name[, default]): to access the attribute of object.
- The hasattr(obj,name): to check if an attribute exists or not.
- The setattr(obj,name,value): to set an attribute. If attribute does not exist, then it would be created.
- The delattr(obj, name): to delete an attribute.

```
hasattr(emp1, 'salary')          # Returns true if 'salary' attribute exists
getattr(emp1, 'salary')          # Returns value of 'salary' attribute
setattr(emp1, 'salary', 7000)    # Set attribute 'age' at 8
delattr(emp1, 'salary')          # Delete attribute 'age'
```

Built-In Class Attributes

Every Python class keeps the following built-in attributes and they can be accessed using dot operator like any other attribute −

- __dict__: Dictionary containing the class's namespace.
- __doc__: Class documentation string or none, if undefined.
- __name__: Class name.
- __module__: Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- __bases__: A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

For the above class let us try to access all these attributes-

```
class Employee:
  'Common base class for all employees'
  empCount = 0

  def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.empCount += 1

  def displayCount(self):
    print ("Total Employee %d" % Employee.empCount)

  def displayEmployee(self):
    print ("Name : ", self.name, ", Salary: ", self.salary)
    emp1 = Employee("Zara", 2000)
    emp2 = Employee("Manni", 5000)

print ("Employee.__doc__:", Employee.__doc__)
print ("Employee.__name__:", Employee.__name__)
```

```
print ("Employee.__module__:", Employee.__module__)
print ("Employee.__bases__:", Employee.__bases__)
print ("Employee.__dict__:", Employee.__dict__ )
```

When the above code is executed, it produces the following result-

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: (,)
Employee.__dict__: {'displayCount': , '__module__': '__main__', '__doc__': 'Common base class for all
employees', 'empCount': 2, '__init__': ,'displayEmployee': , '__weakref__': , '__dict__': }
```

Destroying Objects (Garbage Collection)

Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed as Garbage Collection.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.

An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary). The object's reference count decreases when it is deleted with del, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically.

```
a = 40          # Create object <40>
b = a           # Increase ref. count of <40>
c = [b]         # Increase ref. count of <40>
del a           # Decrease ref. count of <40>
b = 100         # Decrease ref. count of <40>
c[0] = -1       # Decrease ref. count of <40>
```

You normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space. However, a class can implement the special method__del__(), called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non-memory resources used by an instance.

Example

This __del__() destructor prints the class name of an instance that is about to be destroyed.

```
class Point:
  def __init( self, x=0, y=0):
    self.x = x
    self.y = y

  def __del__(self):
    class_name = self.__class__.__name__
    print (class_name, "destroyed")

pt1 = Point()
pt2 = pt1
pt3 = pt1
print (id(pt1), id(pt2), id(pt3) # prints the ids of the objects)
del pt1
del pt2
del pt3
```

When the above code is executed, it produces the following result-

```
3083401324 3083401324 3083401324
Point destroyed
```

Note: Ideally, you should define your classes in a separate file, then you should import them in your main program file using import statement.

In the above example, assuming definition of a Point class is contained in point.py and there is no other executable code in it.

```
import point
p1=point.Point()
```

Class Inheritance

Instead of starting from a scratch, you can create a class by deriving it from a pre-existing class by listing the parent class in parentheses after the new class name.

The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name −

Example

```
class Parent: # define parent class
   parentAttr = 100

   def __init__(self):
      print ("Calling parent constructor")

   def parentMethod(self):
      print ('Calling parent method')

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print ("Parent attribute :", Parent.parentAttr)


class Child(Parent): # define child class

   def __init__(self):
      print ("Calling child constructor")

   def childMethod(self):
      print ('Calling child method')

c = Child() # instance of child
c.childMethod() # child calls its method
c.parentMethod() # calls parent's method
c.setAttr(200) # again call parent's method
c.getAttr() # again call parent's method
```

In a similar way, you can drive a class from multiple parent classes as follows

```
class A: # define your class A
   .....
class B: # define your calss B
   .....
class C(A, B): # subclass of A and B
   .....
```

You can use issubclass() or isinstance() functions to check a relationship of two classes and instances.

- The issubclass(sub, sup) boolean function returns True, if the given subclass sub is indeed a subclass of the superclass sup.
- The isinstance(obj, Class) boolean function returns True, if obj is an instance of class Class or is an instance of a subclass of Class.

Overriding Methods

You can always override your parent class methods. One reason for overriding parent's methods is that you may want special or different functionality in your subclass.

Example

```
class Parent:          # define parent class

  def myMethod(self):
    print ('Calling parent method')
    class Child(Parent):   # define child class

  def myMethod(self):
    print ('Calling child method')

c = Child()            # instance of child
c.myMethod()           # child calls overridden method
```

Data Hiding

An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then will not be directly visible to outsiders.

Example

```
#!/usr/bin/python3
class JustCounter:
  __secretCount = 0

  def count(self):
  self.__secretCount += 1
  print (self.__secretCount)

counter = JustCounter()
counter.count()
counter.count()
print (counter.__secretCount)
```

Handling Exceptions

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them-

- Exception Handling.
- Assertions

Standard Exceptions

Exception Name	Description
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisonError	Raised when division or modulo by zero takes place for all numeric types.

AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
OSError	Raised for operating system-related errors.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

The assert Statement

When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises anAssertionError exception.

```
assert Expression[, Arguments]
```

If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError. AssertionError exceptions can be caught and handled like any other exception, using the try-except statement. If they are not handled, they will terminate the program and produce a traceback.

Example

Here is a function that converts a given temperature from degrees Kelvin to degrees Fahrenheit. Since 0° K is as cold as it gets, the function bails out if it sees a negative temperature −

```
def KelvinToFahrenheit(Temperature):
  assert (Temperature >= 0),"Colder than absolute zero!"
  return ((Temperature-273)*1.8)+32

print (KelvinToFahrenheit(273))
print (int(KelvinToFahrenheit(505.78)))
print (KelvinToFahrenheit(-5))
```

When the above code is executed, it produces the following result-

```
32.0
451

Traceback (most recent call last):
File "test.py", line 9, in
print KelvinToFahrenheit(-5)
```

```
File "test.py", line 4, in KelvinToFahrenheit
assert (Temperature >= 0),"Colder than absolute zero!"
AssertionError: Colder than absolute zero!
```

Handling an Exception

Here is simple syntax of try....except...else blocks

```
try:
   You do your operations here
   ......................

except ExceptionI:
   If there is ExceptionI, then execute this block.

except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Here are few important points about the above-mentioned syntax-

- ✓ A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- ✓ You can also provide a generic except clause, which handles any exception.
- ✓ After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- ✓ The else-block is a good place for code that does not need the try: block's protection.

Example

This example opens a file, writes content in the file and comes out gracefully because there is no problem at all.

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print ("Error: can\'t find file or read data")
else:
   print ("Written content in the file successfully")
   fh.close()
```

Example

This example tries to open a file where you do not have the write permission, so it raises an exception

```
try:
   fh = open("testfile", "r")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print ("Error: can\'t find file or read data")
else:
   print ("Written content in the file successfully")
```

The except Clause with No Exceptions

You can also use the except statement with no exceptions defined as follows

```
try:
   You do your operations here
   ......................
except:
   If there is any exception, then execute this block.
   ......................
```

```
else:
   If there is no exception then execute this block.
```

This kind of a try-except statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

The except Clause with Multiple Exceptions

You can also use the same except statement to handle multiple exceptions as follows

```
try:
   You do your operations here
   ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
   If there is any exception from the given exception list,
   then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

The try-finally Clause

You can use a finally: block along with a try: block. The finally: block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

Note: You can provide except clause(s), or a finally clause, but not both. You cannot use else clause as well along with a finally clause.

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
finally:
   print ("Error: can\'t find file or read data")
   fh.close()
```

If you do not have permission to open the file in writing mode, then this will produce the following result-

```
Error: can't find file or read data
```

Same example can be written more cleanly as follows-

```
try:
   fh = open("testfile", "w")
   try:
      fh.write("This is my test file for exception handling!!")
   finally:
      print ("Going to close the file")
      fh.close()
except IOError:
   print ("Error: can\'t find file or read data")
```

Argument of an Exception

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

Following is an example for a single exception-

```
def temp_convert(var):
try:
  returnint(var)
except ValueError as Argument:
  print("The argument does not contain numbers\n",Argument)

# Call above function here.
temp_convert("xyz")
```

GUI Programming

Tkinter Programming

Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.

Creating a GUI application using Tkinter is an easy task. All you need to do is perform the following steps −

a. Import the Tkinter module.
b. Create the GUI application main window.
c. Add one or more of the above-mentioned widgets to the GUI application.
d. Enter the main event loop to take action against each event triggered by the user.

Example

```
import tkinter    # note that module name has changed from Tkinter in Python 2 to tkinter in Python 3
top = tkinter.Tk()

# Code to add widgets will go here...
top.mainloop()
```

Tkinter Button

The Button widget is used to add buttons in a Python application. These buttons can display text or images that convey the purpose of the buttons. You can attach a function or a method to a button which is called automatically when you click the button.

```
w= Button ( master, option=value, ... )
```

Parameters

- master: This represents the parent window.
- options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
activebackground	Background color when the button is under the cursor.
activeforeground	Foreground color when the button is under the cursor.
bd	Border width in pixels. Default is 2.
bg	Normal background color.
command	Function or method to be called when the button is clicked.
fg	Normal foreground (text) color.
font	Text font to be used for the button's label.
height	Height of the button in text lines (for textual buttons) or pixels(for images).
highlightcolor	The color of the focus highlight when the widget has focus.
image	Image to be displayed on the button (instead of text).
justify	How to show multiple text lines: LEFT to left-justify each line; CENTER to center them; or RIGHT to right-justify.

padx	Additional padding left and right of the text.
pady	Additional padding above and below the text.
relief	Relief specifies the type of the border. Some of the values are SUNKEN, RAISED, GROOVE, and RIDGE.
state	Set this option to DISABLED to gray out the button and make it unresponsive. Has the value ACTIVE when the mouse is over it. Default is NORMAL.
underline	Default is -1, meaning that no character of the text on the button will be underlined. If nonnegative, the corresponding text character will be underlined.
width	Width of the button in letters (if displaying text) or pixels (if displaying an image).
wraplength	If this value is set to a positive number, the text lines will be wrapped to fit within this length.

Methods

Method	**Description**
flash()	Causes the button to flash several times between active and normal colors. Leaves the button in the state it was in originally. Ignored if the button is disabled.
invoke()	Calls the button's callback, and returns what that function returns. Has no effect if the button is disabled or there is no callback.

Example

```
from tkinter import *
from tkinter import messagebox

top = Tk()
top.geometry("100x100")

def helloCallBack():
   msg=messagebox.showinfo( "Hello Python", "Hello World")

B = Button(top, text ="Hello", command = helloCallBack)
B.place(x=50,y=50)
top.mainloop()
```

Tkinter Checkbutton

The Checkbutton widget is used to display a number of options to a user as toggle buttons. The user can then select one or more options by clicking the button corresponding to each option.

```
w = Checkbutton ( master, option, ... )
```

Parameters

- master: This represents the parent window.
- options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	**Description**
activebackground	Background color when the checkbutton is under the cursor.
activeforeground	Foreground color when the checkbutton is under the cursor.
bg	The normal background color displayed behind the label and indicator.
bitmap	To display a monochrome image on a button.
bd	The size of the border around the indicator. Default is 2 pixels.
command	A procedure to be called every time the user changes the state of this checkbutton.
cursor	If you set this option to a cursor name (arrow, dot etc.), the mouse cursor will change to that pattern when it is over the checkbutton.

disabledforeground	The foreground color used to render the text of a disabled checkbutton. The default is a stippled version of the default foreground color.
font	The font used for the text.
fg	The color used to render the text.
height	The number of lines of text on the checkbutton. Default is 1.
highlightcolor	The color of the focus highlight when the checkbutton has the focus.
image	To display a graphic image on the button.
justify	If the text contains multiple lines, this option controls how the text is justified: CENTER, LEFT, or RIGHT.
offvalue	Normally, a checkbutton's associated control variable will be set to 0 when it is cleared (off). You can supply an alternate value for the off state by setting offvalue to that value.
onvalue	Normally, a checkbutton's associated control variable will be set to 1 when it is set (on). You can supply an alternate value for the on state by setting onvalue to that value.
padx	How much space to leave to the left and right of the checkbutton and text. Default is 1 pixel.
pady	How much space to leave above and below the checkbutton and text. Default is 1 pixel.
relief	With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
selectcolor	The color of the checkbutton when it is set. Default is selectcolor="red".
selectimage	If you set this option to an image, that image will appear in the checkbutton when it is set.
state	The default is state=NORMAL, but you can use state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the checkbutton, the state is ACTIVE.
text	The label displayed next to the checkbutton. Use newlines ("\n") to display multiple lines of text.
underline	With the default value of -1, none of the characters of the text label are underlined. Set this option to the index of a character in the text (counting from zero) to underline that character.
variable	The control variable that tracks the current state of the checkbutton. Normally this variable is an IntVar, and 0 means cleared and 1 means set, but see the offvalue and onvalue options above.
width	The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.
wraplength	Normally, lines are not wrapped. You can set this option to a number of characters and all lines will be broken into pieces no longer than that number.

Methods

Method	**Description**
deselect()	Clears (turns off) the checkbutton.
flash()	Flashes the checkbutton a few times between its active and normal colors, but leaves it the way it started.
invoke()	You can call this method to get the same actions that would occur if the user clicked on the checkbutton to change its state.
select()	Sets (turns on) the checkbutton.
toggle()	Clears the checkbutton if set, sets it if cleared.

Example

```
from tkinter import *
import tkinter
top = Tk()
CheckVar1 = IntVar()
CheckVar2 = IntVar()
```

```
C1 = Checkbutton(top, text = "Music", variable = CheckVar1, \
onvalue = 1, offvalue = 0, height=5, width = 20, )

C2 = Checkbutton(top, text = "Video", variable = CheckVar2, \
onvalue = 1, offvalue = 0, height=5, width = 20)

C1.pack()
C2.pack()
top.mainloop()
```

Tkinter Entry

The Entry widget is used to accept single-line text strings from a user.

```
w = Entry( master, option, ... )
```

Parameters

- master: This represents the parent window.
- options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	**Description**
bg	The normal background color displayed behind the label and indicator.
bd	The size of the border around the indicator. Default is 2 pixels.
command	A procedure to be called every time the user changes the state of this checkbutton.
cursor	If you set this option to a cursor name (arrow, dot etc.), the mouse cursor will change to that pattern when it is over the checkbutton.
font	The font used for the text.
exportselection	By default, if you select text within an Entry widget, it is automatically exported to the clipboard. To avoid this exportation, use exportselection=0.
fg	The color used to render the text.
highlightcolor	The color of the focus highlight when the checkbutton has the focus.
justify	If the text contains multiple lines, this option controls how the text is justified: CENTER, LEFT, or RIGHT.
relief	With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
selectbackground	The background color to use displaying selected text.
selectborderwidth	The width of the border to use around selected text. The default is one pixel.
selectforeground	The foreground (text) color of selected text.
show	Normally, the characters that the user types appear in the entry. To make a .password. entry that echoes each character as an asterisk, set show="*".
state	The default is state=NORMAL, but you can use state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the checkbutton, the state is ACTIVE.
textvariable	In order to be able to retrieve the current text from your entry widget, you must set this option to an instance of the StringVar class.
width	The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.
xscrollcommand	If you expect that users will often enter more text than the onscreen size of the widget, you can link your entry widget to a scrollbar.

Methods

Method	Description
delete (first, last=None)	Deletes characters from the widget, starting with the one at index first, up to but not including the character at position last. If the second argument is omitted, only the single character at position first is deleted.
get()	Returns the entry's current text as a string.
icursor (index)	Set the insertion cursor just before the character at the given index.
index (index)	Shift the contents of the entry so that the character at the given index is the leftmost visible character. Has no effect if the text fits entirely within the entry.
insert (index, s)	Inserts string s before the character at the given index.
select_adjust (index)	This method is used to make sure that the selection includes the character at the specified index.
select_clear()	Clears the selection. If there isn't currently a selection, has no effect.
select_from (index)	Sets the ANCHOR index position to the character selected by index, and selects that character.
select_present()	If there is a selection, returns true, else returns false.
select_range (start, end)	Sets the selection under program control. Selects the text starting at the start index, up to but not including the character at the end index. The start position must be before the end position.
select_to (index)	Selects all the text from the ANCHOR position up to but not including the character at the given index.
xview (index)	This method is useful in linking the Entry widget to a horizontal scrollbar.
xview_scroll (number, what)	Used to scroll the entry horizontally. The what argument must be either UNITS, to scroll by character widths, or PAGES, to scroll by chunks the size of the entry widget. The number is positive to scroll left to right, negative to scroll right to left.

```
from tkinter import *
top = Tk()
L1 = Label(top, text="User Name")
L1.pack( side = LEFT)
E1 = Entry(top, bd =5)
E1.pack(side = RIGHT)
top.mainloop()
```

Tkinter Frame

The Frame widget is very important for the process of grouping and organizing other widgets in a somehow friendly way. It works like a container, which is responsible for arranging the position of other widgets. It uses rectangular areas in the screen to organize the layout and to provide padding of these widgets. A frame can also be used as a foundation class to implement complex widgets.

```
w = Frame ( master, option, ... )
```

Parameters

- master: This represents the parent window.
- options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	Description
bg	The normal background color displayed behind the label and indicator.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	If you set this option to a cursor name (arrow, dot etc.), the mouse cursor will change to that pattern when it is over the checkbutton.
height	The vertical dimension of the new frame.

highlightbackground	Color of the focus highlight when the frame does not have focus.
highlightcolor	Color shown in the focus highlight when the frame has the focus.
highlightthickness	Thickness of the focus highlight.
relief	With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
width	The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.

Example

```
from tkinter import *
root = Tk()
frame = Frame(root)
frame.pack()
bottomframe = Frame(root)
bottomframe.pack( side = BOTTOM )

redbutton = Button(frame, text="Red", fg="red")
redbutton.pack( side = LEFT)

greenbutton = Button(frame, text="Brown", fg="brown")
greenbutton.pack( side = LEFT )

bluebutton = Button(frame, text="Blue", fg="blue")
bluebutton.pack( side = LEFT )

blackbutton = Button(bottomframe, text="Black", fg="black")
blackbutton.pack( side = BOTTOM)

root.mainloop()
```

Tkinter Label

This widget implements a display box where you can place text or images. The text displayed by this widget can be updated at any time you want. It is also possible to underline part of the text (like to identify a keyboard shortcut) and
span the text across multiple lines.

```
w = Label ( master, option, ... )
```

Parameters
master: This represents the parent window.
options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	**Description**
anchor	This options controls where the text is positioned if the widget has more space than the text needs. The default is anchor=CENTER, which centers the text in the available space.
bg	The normal background color displayed behind the label and indicator.
bitmap	Set this option equal to a bitmap or image object and the label will display that graphic.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	If you set this option to a cursor name (arrow, dot etc.), the mouse cursor will change to that pattern when it is over the checkbutton. font If you are displaying text in this label (with the text or textvariable option, the font option specifies in what font that text will be displayed.

fg	If you are displaying text or a bitmap in this label, this option specifies the color of the text. If you are displaying a bitmap, this is the color that will appear at the position of the 1-bits in the bitmap.
height	The vertical dimension of the new frame.
image	To display a static image in the label widget, set this option to an image object.
justify	Specifies how multiple lines of text will be aligned with respect to each other: LEFT for flush left, CENTER for centered (the default), or RIGHT for right-justified.
padx	Extra space added to the left and right of the text within the widget. Default is 1.
pady	Extra space added above and below the text within the widget. Default is 1.
relief	Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.
text	To display one or more lines of text in a label widget, set this option to a string containing the text. Internal newlines ("\n") will force a line break.
textvariable	To slave the text displayed in a label widget to a control variable of class StringVar, set this option to that variable.
underline	You can display an underline (_) below the nth letter of the text, counting from 0, by setting this option to n. The default is underline=-1, which means no underlining.
width	Width of the label in characters (not pixels!). If this option is not set, the label will be sized to fit its contents.
wraplength	You can limit the number of characters in each line by setting this option to the desired number. The default value, 0, means that lines will be broken only at newlines.

Example

```
from tkinter import *
root = Tk()
var = StringVar()
label = Label( root, textvariable=var, relief=RAISED )
var.set("Hey!? How are you doing?")
label.pack()
root.mainloop()
```

Tkinter Listbox

The Listbox widget is used to display a list of items from which a user can select a number of items

```
w= Listbox ( master, option, ... )
```

Parameters

- master: This represents the parent window.
- options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	**Description**
bg	The normal background color displayed behind the label and indicator.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	The cursor that appears when the mouse is over the listbox.
font	The font used for the text in the listbox.
fg	The color used for the text in the listbox.
height	Number of lines (not pixels!) shown in the listbox. Default is 10.
highlightcolor	Color shown in the focus highlight when the widget has the focus.
highlightthickness	Thickness of the focus highlight.
relief	Selects three-dimensional border shading effects. The default is SUNKEN.
selectbackground	The background color to use displaying selected text.
selectmode	Determines how many items can be selected, and how mouse drags affect the selection:

	• BROWSE: Normally, you can only select one line out of a listbox. If you click on an item and then drag to a different line, the selection will follow the mouse. This is the default. • SINGLE: You can only select one line, and you can't drag the mouse.wherever you click button 1, that line is selected. • MULTIPLE: You can select any number of lines at once. Clicking on any line toggles whether or not it is selected. • EXTENDED: You can select any adjacent group of lines at once by clicking on the first line and dragging to the last line.
width	The width of the widget in characters. The default is 20.
xscrollcommand	If you want to allow the user to scroll the listbox horizontally, you can link your listbox widget to a horizontal scrollbar.
yscrollcommand	If you want to allow the user to scroll the listbox vertically, you can link your listbox widget to a vertical scrollbar.

Methods

Options	**Description**
activate (index)	Selects the line specifies by the given index.
curselection()	Returns a tuple containing the line numbers of the selected element or elements, counting from 0. If nothing is selected, returns an empty tuple.
delete (first, last=None)	Deletes the lines whose indices are in the range [first, last]. If the second argument is omitted, the single line with index first is deleted.
get (first, last=None)	Returns a tuple containing the text of the lines with indices from first to last, inclusive. If the second argument is omitted, returns the text of the line closest to first.
index (i)	If possible, positions the visible part of the listbox so that the line containing index i is at the top of the widget.
insert (index, *elements)	Insert one or more new lines into the listbox before the line specified by index. Use END as the first argument if you want to add new lines to the end of the listbox.
nearest (y)	Return the index of the visible line closest to the y coordinate y relative to the listbox widget.
see (index)	Adjust the position of the listbox so that the line referred to by index is visible.
size()	Returns the number of lines in the listbox.
xview()	To make the listbox horizontally scrollable, set the command option of the associated horizontal scrollbar to this method.
xview_moveto (fraction)	Scroll the listbox so that the leftmost fraction of the width of its longest line is outside the left side of the listbox. Fraction is in the range [0,1].
xview_scroll (number, what)	Scrolls the listbox horizontally. For the what argument, use either UNITS to scroll by characters, or PAGES to scroll by pages, that is, by the width of the listbox. The number argument tells how many to scroll.
yview()	To make the listbox vertically scrollable, set the command option of the associated vertical scrollbar to this method.
yview_moveto (fraction)	Scroll the listbox so that the top fraction of the width of its longest line is outside the left side of the listbox. Fraction is in the range [0,1].
yview_scroll (number, what)	Scrolls the listbox vertically. For the what argument, use either UNITS to scroll by lines, or PAGES to scroll by pages, that is, by the height of the listbox. The number argument tells how many to scroll.

Example

```
# !/usr/bin/python3
from tkinter import *
import tkinter
```

```
top = Tk()
Lb1 = Listbox(top)
Lb1.insert(1, "Python")
Lb1.insert(2, "Perl")
Lb1.insert(3, "C")
Lb1.insert(4, "PHP")
Lb1.insert(5, "JSP")
Lb1.insert(6, "Ruby")
Lb1.pack()
top.mainloop()
```

Tkinter Menubutton

A menubutton is the part of a drop-down menu that stays on the screen all the time. Every menubutton is associated with a Menu widget that can display the choices for that menubutton when the user clicks on it.

```
w = Menubutton ( master, option, ... )
```

Parameters

- master: This represents the parent window.
- options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	**Description**
activebackground	The background color when the mouse is over the menubutton.
activeforeground	The foreground color when the mouse is over the menubutton.
anchor	This options controls where the text is positioned if the widget has more space than the text needs. The default is anchor=CENTER, which centers the text.
bg	The normal background color displayed behind the label and indicator.
bitmap	To display a bitmap on the menubutton, set this option to a bitmap name.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	The cursor that appears when the mouse is over this menubutton.
direction	Set direction=LEFT to display the menu to the left of the button; use direction=RIGHT to display the menu to the right of the button; or use direction='above' to place the menu above the button.
disabledforeground	The foreground color shown on this menubutton when it is disabled.
fg	The foreground color when the mouse is not over the menubutton.
height	The height of the menubutton in lines of text (not pixels!). The default is to fit the menubutton's size to its contents.
highlightcolor	Color shown in the focus highlight when the widget has the focus.
image	To display an image on this menubutton,
justify	This option controls where the text is located when the text doesn't fill the menubutton: use justify=LEFT to left-justify the text (this is the default); use justify=CENTER to center it, or justify=RIGHT to right-justify.
menu	To associate the menubutton with a set of choices, set this option to the Menu object containing those choices. That menu object must have been created by passing the associated menubutton to the constructor as its first argument.
padx	How much space to leave to the left and right of the text of the menubutton. Default is 1.
pady	How much space to leave above and below the text of the menubutton. Default is 1.
relief	Selects three-dimensional border shading effects. The default is RAISED.
	state Normally, menubuttons respond to the mouse. Set state=DISABLED to gray out the menubutton and make it unresponsive.
text	To display text on the menubutton, set this option to the string containing the desired text. Newlines ("\n") within the string will cause line breaks.

textvariable	You can associate a control variable of class StringVar with this menubutton. Setting that control variable will change the displayed text.
underline	Normally, no underline appears under the text on the menubutton. To underline one of the characters, set this option to the index of that character.
width	The width of the widget in characters. The default is 20.
wraplength	Normally, lines are not wrapped. You can set this option to a number of characters and all lines will be broken into pieces no longer than that number.

Example

```
from tkinter import *
import tkinter
top = Tk()
mb= Menubutton ( top, text="condiments", relief=RAISED )
mb.grid()
mb.menu = Menu ( mb, tearoff = 0 )
mb["menu"] = mb.menu
mayoVar = IntVar()
ketchVar = IntVar()
mb.menu.add_checkbutton ( label="mayo", variable=mayoVar )
mb.menu.add_checkbutton ( label="ketchup", variable=ketchVar )
mb.pack()
top.mainloop()
```

Tkinter Menu

The goal of this widget is to allow us to create all kinds of menus that can be used by our applications. The core functionality provides ways to create three menu types: pop-up, toplevel and pull-down. It is also possible to use other extended widgets to implement new types of menus, such as the OptionMenu widget, which implements a special type that generates a pop-up list of items within a selection.

```
w = Menu ( master, option, ... )
```

Parameters

- master: This represents the parent window.
- options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	**Description**
activebackground	The background color that will appear on a choice when it is under the mouse.
activeborderwidth	Specifies the width of a border drawn around a choice when it is under the mouse. Default is 1 pixel.
activeforeground	The foreground color that will appear on a choice when it is under the mouse.
bg	The background color for choices not under the mouse.
bd	The width of the border around all the choices. Default is 1.
cursor	The cursor that appears when the mouse is over the choices, but only when the menu has been torn off.
disabledforeground	The color of the text for items whose state is DISABLED.
font	The default font for textual choices.
fg	The foreground color used for choices not under the mouse.
postcommand	You can set this option to a procedure, and that procedure will be called every time someone brings up this menu.
relief	The default 3-D effect for menus is relief=RAISED.
image	To display an image on this menubutton.
selectcolor	Specifies the color displayed in checkbuttons and radiobuttons when they are selected.

tearoff	Normally, a menu can be torn off, the first position (position 0) in the list of choices is occupied by the tear-off element, and the additional choices are added starting at position 1. If you set

Methods

Option	Description
add_command (options)	Adds a menu item to the menu.
add_radiobutton(options)	Creates a radio button menu item.
add_checkbutton(options)	Creates a check button menu item.
add_cascade(options)	Creates a new hierarchical menu by associating a given menu to a parent menu
add_separator()	Adds a separator line to the menu.
add(type, options)	Adds a specific type of menu item to the menu.
delete(startindex [, endindex])	Deletes the menu items ranging from startindex to endindex.
entryconfig(index, options)	Allows you to modify a menu item, which is identified by the index, and change its options.
index(item)	Returns the index number of the given menu item label.
insert_separator (index)	Insert a new separator at the position specified by index. tearoff=0, the menu will not have a tear-off feature, and choices will be added starting at position 0.
title	Normally, the title of a tear-off menu window will be the same as the text of the menubutton or cascade that lead to this menu. If you want to change the title of that window, set the title option to that string.
invoke (index)	Calls the command callback associated with the choice at position index. If a checkbutton, its state is toggled between set and cleared; if a radiobutton, that choice is set.
type (index)	Returns the type of the choice specified by index: either "cascade", "checkbutton", "command", "radiobutton", "separator", or "tearoff".

Example

```
from tkinter import *
def donothing():
filewin = Toplevel(root)
button = Button(filewin, text="Do nothing button")
button.pack()
root = Tk()
menubar = Menu(root)
filemenu = Menu(menubar, tearoff=0)
filemenu.add_command(label="New", command=donothing)
filemenu.add_command(label="Open", command=donothing)
filemenu.add_command(label="Save", command=donothing)
filemenu.add_command(label="Save as...", command=donothing)
filemenu.add_command(label="Close", command=donothing)
filemenu.add_separator()
filemenu.add_command(label="Exit", command=root.quit)
menubar.add_cascade(label="File", menu=filemenu)
editmenu = Menu(menubar, tearoff=0)
editmenu.add_command(label="Undo", command=donothing)
editmenu.add_separator()
editmenu.add_command(label="Cut", command=donothing)
editmenu.add_command(label="Copy", command=donothing)
editmenu.add_command(label="Paste", command=donothing)
editmenu.add_command(label="Delete", command=donothing)
editmenu.add_command(label="Select All", command=donothing)
```

```
menubar.add_cascade(label="Edit", menu=editmenu)
helpmenu = Menu(menubar, tearoff=0)
helpmenu.add_command(label="Help Index", command=donothing)
helpmenu.add_command(label="About...", command=donothing)
menubar.add_cascade(label="Help", menu=helpmenu)
root.config(menu=menubar)
root.mainloop()
```

Tkinter Message

This widget provides a multiline and noneditable object that displays texts, automatically breaking lines and justifying their contents. Its functionality is very similar to the one provided by the Label widget, except that it can
also automatically wrap the text, maintaining a given width or aspect ratio.

```
w = Message ( master, option, ... )
```

Parameters

- master: This represents the parent window.
- options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	**Description**
anchor	This options controls where the text is positioned if the widget has more space than the text needs. The default is anchor=CENTER, which centers the text in the available space.
bg	The normal background color displayed behind the label and indicator.
bitmap	Set this option equal to a bitmap or image object and the label will display that graphic.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	If you set this option to a cursor name (arrow, dot etc.), the mouse cursor will change to that pattern when it is over the checkbutton.
font	If you are displaying text in this label (with the text or textvariable option, the font option specifies in what font that text will be displayed.
fg	If you are displaying text or a bitmap in this label, this option specifies the color of the text. If you are displaying a bitmap, this is the color that will appear at the position of the 1-bits in the bitmap.
height	The vertical dimension of the new frame.
image	To display a static image in the label widget, set this option to an image object.
justify	Specifies how multiple lines of text will be aligned with respect to each other: LEFT for flush left, CENTER for centered (the default), or RIGHT for right-justified.
padx	Extra space added to the left and right of the text within the widget. Default is 1.
pady	Extra space added above and below the text within the widget. Default is 1.
relief	Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.
text	To display one or more lines of text in a label widget, set this option to a string containing the text. Internal newlines ("\n") will force a line break.
textvariable	To slave the text displayed in a label widget to a control variable of class StringVar, set this option to that variable.
underline	You can display an underline (_) below the nth letter of the text, counting from 0, by setting this option to n. The default is underline=-1, which means no underlining.
width	Width of the label in characters (not pixels!). If this option is not set, the label will be sized to fit its contents.
wraplength	You can limit the number of characters in each line by setting this option to the desired number. The default value, 0, means that lines will be broken only at newlines.

Example

```
# !/usr/bin/python3
from tkinter import *
root = Tk()
var = StringVar()
label = Message( root, textvariable=var, relief=RAISED )
var.set("Hey!? How are you doing?")
label.pack()
root.mainloop()
```

Tkinter Radiobutton

This widget implements a multiple-choice button, which is a way to offer many possible selections to the user and lets user choose only one of them. In order to implement this functionality, each group of radiobuttons must be associated to the same variable and each one of the buttons must symbolize a single value. You can use the Tab key to switch from one radionbutton to another.

```
w = Radiobutton ( master, option, ... )
```

Parameters

- master: This represents the parent window.
- options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	**Description**
activebackground	The background color when the mouse is over the radiobutton.
activeforeground	The foreground color when the mouse is over the radiobutton.
anchor	If the widget inhabits a space larger than it needs, this option specifies where the radiobutton will sit in that space. The default is anchor=CENTER.
bg	The normal background color behind the indicator and label.
bitmap	To display a monochrome image on a radiobutton, set this option to a bitmap.
borderwidth	The size of the border around the indicator part itself. Default is 2 pixels.
command	A procedure to be called every time the user changes the state of this radiobutton.
cursor	If you set this option to a cursor name (arrow, dot etc.), the mouse cursor will change to that pattern when it is over the radiobutton.
font	The font used for the text.
fg	The color used to render the text.
height	The number of lines (not pixels) of text on the radiobutton. Default is 1.
highlightbackground	The color of the focus highlight when the radiobutton does not have focus.
highlightcolor	The color of the focus highlight when the radiobutton has the focus.
image	To display a graphic image instead of text for this radiobutton, set this option to an image object.
justify	If the text contains multiple lines, this option controls how the text is justified: CENTER (the default), LEFT, or RIGHT.
padx	How much space to leave to the left and right of the radiobutton and text. Default is 1.
pady	How much space to leave above and below the radiobutton and text. Default is 1.
relief	Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.
selectcolor	The color of the radiobutton when it is set. Default is red.
selectimage	If you are using the image option to display a graphic instead of text when the radiobutton is cleared, you can set the selectimage option to a different image that will be displayed when the radiobutton is set.

state	The default is state=NORMAL, but you can set state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the radiobutton, the state is ACTIVE.
text of text.	The label displayed next to the radiobutton. Use newlines ("\n") to display multiple lines
textvariable	To slave the text displayed in a label widget to a control variable of class StringVar, set this option to that variable.
underline	You can display an underline (_) below the nth letter of the text, counting from 0, by setting this option to n. The default is underline=-1, which means no underlining.
value	When a radiobutton is turned on by the user, its control variable is set to its current value option. If the control variable is an IntVar, give each radiobutton in the group a different integer value option. If the control variable is aStringVar, give each radiobutton a different string value option.
variable	The control variable that this radiobutton shares with the other radiobuttons in the group. This can be either an IntVar or a StringVar.
width	Width of the label in characters (not pixels!). If this option is not set, the label will be sized to fit its contents.
wraplength	You can limit the number of characters in each line by setting this option to the desired number. The default value, 0, means that lines will be broken only at newlines.

Methods

Methods	**Description**
deselect()	Clears (turns off) the radiobutton.
flash()	Flashes the radiobutton a few times between its active and normal colors, but leaves it the way it started.
invoke()	You can call this method to get the same actions that would occur if the user clicked on the radiobutton to change its state.
select()	Sets (turns on) the radiobutton.

Example

```
from tkinter import *
def sel():
selection = "You selected the option " + str(var.get())
label.config(text = selection)
root = Tk()
var = IntVar()

R1 = Radiobutton(root, text="Option 1", variable=var, value=1,command=sel)
R1.pack( anchor = W )

R2 = Radiobutton(root, text="Option 2", variable=var, value=2,command=sel)
R2.pack( anchor = W )

R3 = Radiobutton(root, text="Option 3", variable=var, value=3,command=sel)
R3.pack( anchor = W)

label = Label(root)
label.pack()
root.mainloop()
```

Tkinter Scale

The Scale widget provides a graphical slider object that allows you to select values from a specific scale.
w = Scale (master, option, ...)

Parameters

- master: This represents the parent window.
- options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	**Description**
activebackground	The background color when the mouse is over the scale.
bg	The background color of the parts of the widget that are outside the trough.
bd	Width of the 3-d border around the trough and slider. Default is 2 pixels.
command	A procedure to be called every time the slider is moved. This procedure will be passed one argument, the new scale value. If the slider is moved rapidly, you may not get a callback for every possible position, but you'll certainly get a callback when it settles.
cursor	If you set this option to a cursor name (arrow, dot etc.), the mouse cursor will change to that pattern when it is over the scale.
digits	The way your program reads the current value shown in a scale widget is through a control variable. The control variable for a scale can be an IntVar, a DoubleVar (float), or a StringVar. If it is a string variable, the digits option controls how many digits to use when the numeric scale value is converted to a string.
font	The font used for the label and annotations.
fg	The color of the text used for the label and annotations.
from_	A float or integer value that defines one end of the scale's range.
highlightbackground	The color of the focus highlight when the scale does not have focus.
highlightcolor	The color of the focus highlight when the scale has the focus.
label	You can display a label within the scale widget by setting this option to the label's text. The label appears in the top left corner if the scale is horizontal, or the top right corner if vertical. The default is no label.
length	The length of the scale widget. This is the x dimension if the scale is horizontal, or the y dimension if vertical. The default is 100 pixels.
orient	Set orient=HORIZONTAL if you want the scale to run along the x dimension, or orient=VERTICAL to run parallel to the y-axis. Default is horizontal.
relief	Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.
repeatdelay	This option controls how long button 1 has to be held down in the trough before the slider starts moving in that direction repeatedly. Default is repeatdelay=300, and the units are milliseconds.
resolution	Normally, the user will only be able to change the scale in whole units. Set this option to some other value to change the smallest increment of the scale's value. For example, if from_=-1.0 and to=1.0, and you set resolution=0.5, the scale will have 5 possible values: -1.0, -0.5, 0.0, +0.5, and +1.0.
showvalue	Normally, the current value of the scale is displayed in text form by the slider (above it for horizontal scales, to the left for vertical scales). Set this option to 0 to suppress that label.
sliderlength	Normally the slider is 30 pixels along the length of the scale. You can change that length by setting the sliderlength option to your desired length.
state	Normally, scale widgets respond to mouse events, and when they have the focus, also keyboard events. Set state=DISABLED to make the widget unresponsive.
takefocus	Normally, the focus will cycle through scale widgets. Set this option to 0 if you don't want this behavior.

tickinterval	To display periodic scale values, set this option to a number, and ticks will be displayed on multiples of that value. For example, if from_=0.0, to=1.0, and tickinterval=0.25, labels will be displayed along the scale at values 0.0, 0.25, 0.50, 0.75, and 1.00. These labels appear below the scale if horizontal, to its left if vertical. Default is 0, which suppresses display of ticks.
to	A float or integer value that defines one end of the scale's range; the other end is defined by the from_ option, discussed above. The to value can be either greater than or less than the from_ value. For vertical scales, the to value defines the bottom of the scale; for horizontal scales, the right end.
troughcolor	The color of the trough.
variable	The control variable for this scale, if any. Control variables may be from class IntVar, DoubleVar (float), or StringVar. In the latter case, the numerical value will be converted to a string.
width	The width of the trough part of the widget. This is the x dimension for vertical scales and the y dimension if the scale has orient=HORIZONTAL. Default is 15 pixels.

Methods

Methods	**Description**
get()	This method returns the current value of the scale.
set (value)	Sets the scale's value.

Example

```
from tkinter import *
def sel():
selection = "Value = " + str(var.get())
label.config(text = selection)
root = Tk()
var = DoubleVar()
scale = Scale( root, variable = var )
scale.pack(anchor=CENTER)
button = Button(root, text="Get Scale Value", command=sel)
button.pack(anchor=CENTER)
label = Label(root)
label.pack()
root.mainloop()
```

Tkinter Scrollbar

This widget provides a slide controller that is used to implement vertical scrolled widgets, such as Listbox, Text and Canvas. Note that you can also create horizontal scrollbars on Entry widgets.

```
w = Scrollbar ( master, option, ... )
```

Parameters

- master: This represents the parent window.
- options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	**Description**
activebackground	The color of the slider and arrowheads when the mouse is over them.
bg	The color of the slider and arrowheads when the mouse is not over them.
bd	The width of the 3-d borders around the entire perimeter of the trough, and also the width of the 3-d effects on the arrowheads and slider. Default is no border around the trough, and a 2-pixel border around the arrowheads and slider.

command	A procedure to be called whenever the scrollbar is moved.
cursor	The cursor that appears when the mouse is over the scrollbar.
elementborderwidth	The width of the borders around the arrowheads and slider. The default is elementborderwidth=-1, which means to use the value of the borderwidth option.
highlightbackground	The color of the focus highlight when the scrollbar does not have focus.
highlightcolor	The color of the focus highlight when the scrollbar has the focus.
highlightthickness	The thickness of the focus highlight. Default is 1. Set to 0 to suppress display of the focus highlight.
jump	This option controls what happens when a user drags the slider. Normally (jump=0), every small drag of the slider causes the command callback to be called. If you set this option to 1, the callback isn't called until the user releases the mouse button.
orient	Set orient=HORIZONTAL for a horizontal scrollbar, orient=VERTICAL for a vertical one.
repeatdelay	This option controls how long button 1 has to be held down in the trough before the slider starts moving in that direction repeatedly. Default is repeatdelay=300, and the units are milliseconds.
repeatinterval	repeatinterval
takefocus	Normally, you can tab the focus through a scrollbar widget. Set takefocus=0 if you don't want this behavior.
troughcolor	The color of the trough.
width	Width of the scrollbar (its y dimension if horizontal, and its x dimension if vertical).

Default is 16.

Methods

Methods	**Description**
get()	Returns two numbers (a, b) describing the current position of the slider. The a value gives the position of the left or top edge of the slider, for horizontal and vertical scrollbars respectively; the b value gives the position of the right or bottom edge.
set (first, last)	To connect a scrollbar to another widget w, set w's xscrollcommand or yscrollcommand to the scrollbar's set() method. The arguments have the same meaning as the values returned by the get() method.

Example

```
from tkinter import *
root = Tk()
scrollbar = Scrollbar(root)
scrollbar.pack( side = RIGHT, fill=Y )
mylist = Listbox(root, yscrollcommand = scrollbar.set )
for line in range(100):
mylist.insert(END, "This is line number " + str(line))
mylist.pack( side = LEFT, fill = BOTH )
scrollbar.config( command = mylist.yview )
mainloop()
```

Tkinter Text

Text widgets provide advanced capabilities that allow you to edit a multiline text and format the way it has to be displayed, such as changing its color and font. You can also use elegant structures like tabs and marks to locate specific sections of the text, and apply changes to those areas. Moreover, you can embed windows and images in the text because this widget was designed to handle both plain and formatted text.

```
w = Text ( master, option, ... )
```

Parameters

- master: This represents the parent window.
- options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	**Description**
bg	The default background color of the text widget.
bd	The width of the border around the text widget. Default is 2 pixels.
cursor	The cursor that will appear when the mouse is over the text widget.
exportselection	Normally, text selected within a text widget is exported to be the selection in the window manager. Set exportselection=0 if you don't want that behavior.
font	The default font for text inserted into the widget.
fg	The color used for text (and bitmaps) within the widget. You can change the color for tagged regions; this option is just the default.
height	The height of the widget in lines (not pixels!), measured according to the current font size.
highlightbackground	The color of the focus highlight when the text widget does not have focus.
highlightcolor	The color of the focus highlight when the text widget has the focus.
highlightthickness	The thickness of the focus highlight. Default is 1. Set highlightthickness=0 to suppress display of the focus highlight.
insertbackground	The color of the insertion cursor. Default is black.
insertborderwidth	Size of the 3-D border around the insertion cursor. Default is 0.
insertofftime	The number of milliseconds the insertion cursor is off during its blink cycle. Set this option to zero to suppress blinking. Default is 300.
insertontime	The number of milliseconds the insertion cursor is on during its blink cycle. Default is 600.
insertwidth	Width of the insertion cursor (its height is determined by the tallest item in its line). Default is 2 pixels.
padx	The size of the internal padding added to the left and right of the text area. Default is one pixel.
pady	The size of the internal padding added above and below the text area. Default is one pixel.
relief	The 3-D appearance of the text widget. Default is relief=SUNKEN.
selectbackground	The background color to use displaying selected text.
selectborderwidth	The width of the border to use around selected text.
spacing1	This option specifies how much extra vertical space is put above each line of text. If a line wraps, this space is added only before the first line it occupies on the display. Default is 0.
spacing2	This option specifies how much extra vertical space to add between displayed lines of text when a logical line wraps. Default is 0.
spacing3	This option specifies how much extra vertical space is added below each line of text. If a line wraps, this space is added only after the last line it occupies on the display. Default is 0.
state	Normally, text widgets respond to keyboard and mouse events; set state=NORMAL to get this behavior. If you set state=DISABLED, the text widget will not respond, and you won't be able to modify its contents programmatically either.
tabs	This option controls how tab characters position text.
width	The width of the widget in characters (not pixels!), measured according to the current font size.
wrap	This option controls the display of lines that are too wide. Set wrap=WORD and it will break the line after the last word that will fit. With the default behavior, wrap=CHAR, any line that gets too long will be broken at any character.
xscrollcommand	To make the text widget horizontally scrollable, set this option to the set() method of the horizontal scrollbar.

yscrollcommand	To make the text widget vertically scrollable, set this option to the set() method of the vertical scrollbar.

Methods

Methods	**Description**
delete(startindex [,endindex])	This method deletes a specific character or a range of text.
get(startindex [,endindex])	This method returns a specific character or a range of text.
index(index)	Returns the absolute value of an index based on the given index.
insert(index [,string]...)	This method inserts strings at the specified index location.
see(index)	This method returns true if the text located at the index position is visible.

Text widgets support three distinct helper structures: **Marks**, **Tabs**, and **Indexes**-

Marks are used to bookmark positions between two characters within a given text. We have the following methods available when handling marks:

Methods	**Description**
index(mark)	Returns the line and column location of a specific mark.
mark_gravity(mark [,gravity])	Returns the gravity of the given mark. If the second argument is provided, the gravity is set for the given mark.
mark_names()	Returns all marks from the Text widget.
mark_set(mark, index)	Informs a new position to the given mark.
mark_unset(mark)	Removes the given mark from the Text widget.

Tags are used to associate names to regions of text which makes easy the task of modifying the display settings of specific text areas. Tags are also used to bind event callbacks to specific ranges of text. Following are the available methods for handling tabs-

Methods	**Description**
tag_add(tagname, startindex[,endindex] ...)	This method tags either the position defined by startindex, or a range delimited by the positions startindex and endindex.
tag_config	You can use this method to configure the tag properties, which include, justify(center, left, or right), tabs(this property has the same functionality of the Text widget tabs's property), and underline(used to underline the tagged text).
tag_delete(tagname)	This method is used to delete and remove a given tag.
tag_remove(tagname [,startindex[.endindex]] ...)	After applying this method, the given tag is removed from the provided area without deleting the actual tag definition.

Example

```
from tkinter import *
root = Tk()
text = Text(root)
text.insert(INSERT, "Hello.....")
text.insert(END, "Bye Bye.....")
text.pack()
text.tag_add("here", "1.0", "1.4")
text.tag_add("start", "1.8", "1.13")
text.tag_config("here", background="yellow", foreground="blue")
text.tag_config("start", background="black", foreground="green")
root.mainloop()
```

Tkinter Toplevel

Toplevel widgets work as windows that are directly managed by the window manager. They do not necessarily have a parent widget on top of them. Your application can use any number of top-level windows.

```
w = Toplevel ( option, ... )
```

Parameters

- options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	**Description**
bg	The background color of the window.
bd	Border width in pixels; default is 0.
cursor	The cursor that appears when the mouse is in this window.
class_	Normally, text selected within a text widget is exported to be the selection in the window manager. Set exportselection=0 if you don't want that behavior.
font	The default font for text inserted into the widget.
fg	The color used for text (and bitmaps) within the widget. You can change the color for tagged regions; this option is just the default.
height	Window height.
relief	Normally, a top-level window will have no 3-d borders around it. To get a shaded border, set the bd option larger that its default value of zero, and set the relief option to one of the constants.
width	The desired width of the window.

Methods

Methods	**Description**
deiconify()	Displays the window, after using either the iconify or the withdraw methods.
frame()	Returns a system-specific window identifier.
group(window)	Adds the window to the window group administered by the given window.
iconify()	Turns the window into an icon, without destroying it.
protocol(name, function)	Registers a function as a callback which will be called for the given protocol.
state()	Returns the current state of the window. Possible values are normal, iconic, withdrawn and icon.
transient([master])	Turns the window into a temporary(transient) window for the given master or to the window's parent, when no argument is given.
withdraw()	Removes the window from the screen, without destroying it.
maxsize(width, height)	Defines the maximum size for this window.
minsize(width, height)	Defines the minimum size for this window.
positionfrom(who)	Defines the position controller.
resizable(width, height)	Defines the resize flags, which control whether the window can be resized.
sizefrom(who)	Defines the size controller.
title(string)	Defines the window title.

Example

```
from tkinter import *
root = Tk()
root.title("hello")
top = Toplevel()
top.title("Python")
top.mainloop()
```

Tkinter Spinbox

The Spinbox widget is a variant of the standard Tkinter Entry widget, which can be used to select from a fixed number of values.

```
w = Spinbox( master, option, ... )
```

Parameters

- master: This represents the parent window.
- options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	**Description**
activebackground	The color of the slider and arrowheads when the mouse is over them.
bg	The color of the slider and arrowheads when the mouse is not over them.
bd	The width of the 3-d borders around the entire perimeter of the trough, and also the width of the 3-d effects on the arrowheads and slider. Default is no border around the trough, and a 2-pixel border around the arrowheads and slider.
command	A procedure to be called whenever the scrollbar is moved.
cursor	The cursor that appears when the mouse is over the scrollbar.
disabledbackground	The background color to use when the widget is disabled.
disabledforeground	The text color to use when the widget is disabled.
fg	Text color.
font	The font to use in this widget.
format	Format string. No default value.
from_	The minimum value. Used together with to to limit the spinbox range.
justify	Default is LEFT
relief	Default is SUNKEN.
repeatdelay	Together with repeatinterval, this option controls button autorepeat. Both values are given in milliseconds.
repeatinterval	See repeatdelay.
state	One of NORMAL, DISABLED, or "readonly". Default is NORMAL.
textvariable	No default value.
to	See from.
validate	Validation mode. Default is NONE.
validatecommand	Validation callback. No default value.
values	A tuple containing valid values for this widget. Overrides from/to/increment.
vcmd	Same as validatecommand.
width	Widget width, in character units. Default is 20.
wrap	If true, the up and down buttons will wrap around.
xscrollcommand	Used to connect a spinbox field to a horizontal scrollbar. This option should be set to the set method of the corresponding scrollbar.

Methods

Methods	**Description**
delete(startindex [,endindex])	This method deletes a specific character or a range of text.
get(startindex [,endindex])	This method returns a specific character or a range of text.
identify(x, y)	Identifies the widget element at the given location.
index(index)	Returns the absolute value of an index based on the given index.
insert(index [,string]...)	This method inserts strings at the specified index location.
invoke(element)	Invokes a spinbox button.

Example

```
from tkinter import *
```

```
master = Tk()
w = Spinbox(master, from_=0, to=10)
w.pack()
mainloop()
```

Tkinter PanedWindow

A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically. Each pane contains one widget and each pair of panes is separated by a moveable (via mouse movements) sash. Moving a sash causes the widgets on either side of the sash to be resized.

```
w = PanedWindow( master, option, ... )
```

Parameters

- master: This represents the parent window.
- options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	**Description**
bg	The color of the slider and arrowheads when the mouse is not over them.
bd	The width of the 3-d borders around the entire perimeter of the trough, and also the width of the 3-d effects on the arrowheads and slider. Default is no border around the trough, and a 2-pixel border around the arrowheads and slider.
borderwidth	Default is 2.
cursor	The cursor that appears when the mouse is over the window.
handlepad	Default is 8.
handlesize	Default is 8.
height	No default value.
orient	Default is HORIZONTAL.
relief	Default is FLAT.
sashcursor	No default value.
sashrelief	Default is RAISED.
sashwidth	Default is 2.
showhandle	No default value
width	No default value.

Methods

Methods	**Description**
add(child, options)	Adds a child window to the paned window.
get(startindex [,endindex])	This method returns a specific character or a range of text.
config(options)	Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

Example

```
#create a 3-pane widget-
from tkinter import *
m1 = PanedWindow()
m1.pack(fill=BOTH, expand=1)
left = Entry(m1, bd=5)
m1.add(left)
m2 = PanedWindow(m1, orient=VERTICAL)
m1.add(m2)
top = Scale( m2, orient=HORIZONTAL)
m2.add(top)
bottom = Button(m2, text="OK")
```

```
m2.add(bottom)
mainloop()
```

Tkinter LabelFrame

A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts. This widget has the features of a frame plus the ability to display a label.

```
w = LabelFrame( master, option, ... )
```

Parameters

- master: This represents the parent window.
- options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	**Description**
bg	The normal background color displayed behind the label and indicator.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	If you set this option to a cursor name (arrow, dot etc.), the mouse cursor will change to that pattern when it is over the checkbutton.
font	The vertical dimension of the new frame.
height	The vertical dimension of the new frame.
labelAnchor	Specifies where to place the label.
highlightbackground	Color of the focus highlight when the frame does not have focus.
highlightcolor	Color shown in the focus highlight when the frame has the focus.
highlightthickness	Thickness of the focus highlight.
relief	With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
text	Specifies a string to be displayed inside the widget.
width	Specifies the desired width for the window.

Example

```
from tkinter import *
root = Tk()
labelframe = LabelFrame(root, text="This is a LabelFrame")
labelframe.pack(fill="both", expand="yes")
left = Label(labelframe, text="Inside the LabelFrame")
left.pack()
root.mainloop()
```

Tkinter tkMessageBox

The tkMessageBox module is used to display message boxes in your applications. This module provides a number of functions that you can use to display an appropriate message. Some of these functions are showinfo, showwarning, showerror, askquestion, askokcancel, askyesno, and askretryignore.

```
tkMessageBox.FunctionName(title, message [, options])
```

Parameters

- FunctionName: This is the name of the appropriate message box function.
- title: This is the text to be displayed in the title bar of a message box.
- message: This is the text to be displayed as a message.
- options: options are alternative choices that you may use to tailor a standard essage box. Some of the options that you can use are default and parent. The default option is used to specify the default button, such as ABORT, RETRY, or IGNORE in the message box. The parent option is used to specify the window on top of which the message box is to be displayed.

You could use one of the following functions with dialogue box-

- showinfo()
- showwarning()
- showerror ()
- askquestion()
- askokcancel()
- askyesno ()
- askretrycancel ()

Example

```
from tkinter import *
from tkinter import messagebox
top = Tk()
top.geometry("100x100")
def hello():
messagebox.showinfo("Say Hello", "Hello World")
B1 = Button(top, text = "Say Hello", command = hello)
B1.place(x=35,y=50)
top.mainloop()
```

Standard Attributes

Let us look at how some of the common attributes, such as sizes, colors and fonts are specified.

Tkinter Dimensions

Various lengths, widths, and other dimensions of widgets can be described in many different units.

- ✓ If you set a dimension to an integer, it is assumed to be in pixels.
- ✓ You can specify units by setting a dimension to a string containing a number followed by.

Character	**Description**
c	Centimeters
i	Inches
m	Millimeters
p	Printer's points (about 1/72")

Length options

Tkinter expresses a length as an integer number of pixels. Here is the list of common length options-

- borderwidth: Width of the border which gives a three-dimensional look to the widget.
- highlightthickness: Width of the highlight rectangle when the widget has focus .
- padX padY: Extra space the widget requests from its layout manager beyond the minimum the widget needs to display its contents in the x and y directions.
- selectborderwidth: Width of the three-dimentional border around selected items of the widget.
- wraplength: Maximum line length for widgets that perform word wrapping.
- height: Desired height of the widget; must be greater than or equal to 1.
- underline: Index of the character to underline in the widget's text (0 is the first character, 1 the second one, and so on).
- width: Desired width of the widget.

Tkinter Colors

Tkinter represents colors with strings. There are two general ways to specify colors in Tkinter-

- ✓ You can use a string specifying the proportion of red, green and blue in hexadecimal digits. For example, "#fff" is white, "#000000" is black, "#000fff000" is pure green, and "#00ffff" is pure cyan (green plus blue).

- ✓ You can also use any locally defined standard color name. The colors "white", "black", "red", "green", "blue", "cyan", "yellow", and "magenta" will always be available.

Color options

The common color options are-

- activebackground: Background color for the widget when the widget is active.
- activeforeground: Foreground color for the widget when the widget is active.
- background: Background color for the widget. This can also be represented as bg.
- disabledforeground: Foreground color for the widget when the widget is disabled.
- foreground: Foreground color for the widget. This can also be represented as fg.
- highlightbackground: Background color of the highlight region when the widget has focus.
- highlightcolor: Foreground color of the highlight region when the widget has focus.
- selectbackground: Background color for the selected items of the widget.
- selectforeground: Foreground color for the selected items of the widget.

Tkinter Fonts

Simple Tuple Fonts

As a tuple whose first element is the font family, followed by a size in points, optionally followed by a string containing one or more of the style modifiers bold, italic, underline and overstrike.

Example

- ("Helvetica", "16") for a 16-point Helvetica regular.
- ("Times", "24", "bold italic") for a 24-point Times bold italic.

Font object Fonts

You can create a "font object" by importing the tkFont module and using its Font class constructor −

```
import tkFont
font = tkFont.Font ( option, ... )
```

Here is the list of options

- family: The font family name as a string.
- size: The font height as an integer in points. To get a font n pixels high, use -n.
- weight: "bold" for boldface, "normal" for regular weight.
- slant: "italic" for italic, "roman" for unslanted.
- underline: 1 for underlined text, 0 for normal.
- overstrike: 1 for overstruck text, 0 for normal.

Example

```
helv36 = tkFont.Font(family="Helvetica",size=36,weight="bold")
```

Geometry Management

All Tkinter widgets have access to the specific geometry management methods, which have the purpose of organizing widgets throughout the parent widget area. Tkinter exposes the following geometry manager classes: pack, grid, and place.

Tkinter pack() Method

This geometry manager organizes widgets in blocks before placing them in the parent widget.

```
widget.pack( pack_options )
```

Here is the list of possible options-

- expand: When set to true, widget expands to fill any space not otherwise used in widget's parent.
- fill: Determines whether widget fills any extra space allocated to it by the packer, or keeps its own minimal dimensions: NONE (default), X (fill only horizontally), Y (fill only vertically), or BOTH (fill both horizontally and vertically).
- side: Determines which side of the parent widget packs against: TOP (default), BOTTOM, LEFT, or RIGHT.

Example

Try the following example by moving cursor on different buttons-

```
from tkinter import *
root = Tk()
frame = Frame(root)
frame.pack()
bottomframe = Frame(root)
bottomframe.pack( side = BOTTOM )
redbutton = Button(frame, text="Red", fg="red")
redbutton.pack( side = LEFT)
greenbutton = Button(frame, text="Brown", fg="brown")
greenbutton.pack( side = LEFT )
bluebutton = Button(frame, text="Blue", fg="blue")
bluebutton.pack( side = LEFT )
blackbutton = Button(bottomframe, text="Black", fg="black")
blackbutton.pack( side = BOTTOM)
root.mainloop()
```

Tkinter grid() Method

This geometry manager organizes widgets in a table-like structure in the parent widget.

```
widget.grid( grid_options )
```

Here is the list of possible options-

- column : The column to put widget in; default 0 (leftmost column).
- columnspan: How many columns widgetoccupies; default 1.
- ipadx, ipady :How many pixels to pad widget, horizontally and vertically, inside widget's borders.
- padx, pady : How many pixels to pad widget, horizontally and vertically, outside v's borders.
- row: The row to put widget in; default the first row that is still empty.
- rowspan : How many rowswidget occupies; default 1.
- sticky : What to do if the cell is larger than widget. By default, with sticky='', widget is centered in its cell. sticky may be the string concatenation of zero or more of N, E, S, W, NE, NW, SE, and SW, compass directions indicating the sides and corners of the cell to which widget sticks.

Example

```
from tkinter import *
root = Tk( )
b=0
for r in range(6):
for c in range(6):
b=b+1
Button(root, text=str(b),
borderwidth=1 ).grid(row=r,column=c)
root.mainloop()
```

Tkinter place() Method

This geometry manager organizes widgets by placing them in a specific position in the parent widget.

```
widget.place( place_options )
```

Here is the list of possible options-

- anchor : The exact spot of widget other options refer to: may be N, E, S, W, NE, NW, SE, or SW, compass directions indicating the corners and sides of widget; default is NW (the upper left corner of widget)
- bordermode : INSIDE (the default) to indicate that other options refer to the parent's inside (ignoring the parent's border); OUTSIDE otherwise.
- height, width : Height and width in pixels.
- relheight, relwidth : Height and width as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget.
- relx, rely : Horizontal and vertical offset as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget.
- x, y : Horizontal and vertical offset in pixels.

Example

Try the following example by moving cursor on different buttons-

```
from tkinter import *
top = Tk()
L1 = Label(top, text="Physics")
L1.place(x=10,y=10)
E1 = Entry(top, bd =5)
E1.place(x=60,y=10)
L2=Label(top,text="Maths")
L2.place(x=10,y=50)
E2=Entry(top,bd=5)
E2.place(x=60,y=50)
L3=Label(top,text="Total")
L3.place(x=10,y=150)
E3=Entry(top,bd=5)
E3.place(x=60,y=150)
B = Button(top, text ="Add")
B.place(x=100, y=100)
top.geometry("250x250+10+10")
top.mainloop()
```

Database Access (MySQL)

Database connection prep:

a. Install xampp
b. Set mysql database, add table and add records
c. Check if you have PyMySql module
d. If it produces error, install pymysql

```
pip install pymysql
pip3 install pymysql
```

e. Test connect and fetch records

```
import pymysql

db=pymysql.connect('localhost','root','','hpdatabase')
cursor=db.cursor();

sql='select * from employees'
```

```
try:
        cursor.execute(sql)
        results=cursor.fetchall()

        for row in results:
                print(str(row[0]) + "\t" + str(row[1]) + "\t" + str(row[2]))
        print('done.')
except ValueError as e:
        print('[error]'+e)

db.close()
```

Add new record:

```
import pymysql
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )
cursor = db.cursor()

sql = "INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX, INCOME) \
VALUES ('%s', '%s', '%d', '%c', '%d' )" % ('Mac', 'Mohan', 20, 'M', 2000)

try:
  cursor.execute(sql)
  db.commit()

except:
  db.rollback()

db.close()
```

Delete record:

```
#!/usr/bin/python3
import pymysql

db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )
cursor = db.cursor()
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)

try:
  cursor.execute(sql)
  db.commit()

except:
  db.rollback()

db.close()
```

Update record

```
import pymysql

db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )
cursor = db.cursor()
sql = "UPDATE EMPLOYEE SET AGE = AGE + 1 WHERE SEX = '%c'" % ('M')
```

```
try:
  cursor.execute(sql)
  db.commit()

except:
  db.rollback()

db.close()
```

Module 2: Python and MySQL
Creating Database and Table

```
import pymysql
from pymysql import Error

def create_database_and_table():
   try:
      # Connect to the MySQL server
      connection = pymysql.connect(
         host='localhost',
         user='admin',
         password='123'
      )

      cursor = connection.cursor()

      # Create database if it does not exist
      cursor.execute("CREATE DATABASE IF NOT EXISTS database1")
      cursor.execute("USE database1")

      # Create table if it does not exist
      create_table_query = """
      CREATE TABLE IF NOT EXISTS sales (
         salesid INT AUTO_INCREMENT PRIMARY KEY,
         productname VARCHAR(255),
         quantity INT,
         transactiondate DATETIME,
         branch VARCHAR(255)
      )
      """
      cursor.execute(create_table_query)
      connection.commit()
      print("Database and table created successfully")

   except Error as e:
      print(f"Error: {e}")

   finally:
      if connection:
         cursor.close()
         connection.close()
         print("MySQL connection is closed")

if __name__ == "__main__":
   create_database_and_table()
```

Manipulating Database (Insert, Select, Update, Delete, Where Statement, Order by)

```
import pymysql
from pymysql import Error

def connect():
    """ Connect to the MySQL database """
    try:
        connection = pymysql.connect(
            host='localhost',
            user='admin',
            password='123',
            database='database1'
        )
        return connection
    except Error as e:
        print(f"Error: {e}")
        return None

def create_database_and_table():
    connection = connect()
    if connection:
        try:
            cursor = connection.cursor()
            create_table_query = """
            CREATE TABLE IF NOT EXISTS sales (
                salesid INT AUTO_INCREMENT PRIMARY KEY,
                productname VARCHAR(255),
                quantity INT,
                transactiondate DATETIME,
                branch VARCHAR(255)
            )
            """
            cursor.execute(create_table_query)
            connection.commit()
            print("Database and table created successfully")
        except Error as e:
            print(f"Error: {e}")
        finally:
            cursor.close()
            connection.close()

def create_sale(productname, quantity, transactiondate, branch):
    connection = connect()
    if connection:
        try:
            cursor = connection.cursor()
            insert_query = """
            INSERT INTO sales (productname, quantity, transactiondate, branch)
            VALUES (%s, %s, %s, %s)
            """
            cursor.execute(insert_query, (productname, quantity, transactiondate, branch))
            connection.commit()
            print("Sale record created successfully")
        except Error as e:
            print(f"Error: {e}")
        finally:
            cursor.close()
            connection.close()
```

```
def read_sales():
    connection = connect()
    if connection:
        try:
            cursor = connection.cursor()
            cursor.execute("SELECT * FROM sales")
            rows = cursor.fetchall()
            for row in rows:
                print(row)
        except Error as e:
            print(f"Error: {e}")
        finally:
            cursor.close()
            connection.close()

def update_sale(salesid, productname, quantity, transactiondate, branch):
    connection = connect()
    if connection:
        try:
            cursor = connection.cursor()
            update_query = """
            UPDATE sales
            SET productname = %s, quantity = %s, transactiondate = %s, branch = %s
            WHERE salesid = %s
            """
            cursor.execute(update_query, (productname, quantity, transactiondate, branch, salesid))
            connection.commit()
            print("Sale record updated successfully")
        except Error as e:
            print(f"Error: {e}")
        finally:
            cursor.close()
            connection.close()

def delete_sale(salesid):
    connection = connect()
    if connection:
        try:
            cursor = connection.cursor()
            delete_query = "DELETE FROM sales WHERE salesid = %s"
            cursor.execute(delete_query, (salesid,))
            connection.commit()
            print("Sale record deleted successfully")
        except Error as e:
            print(f"Error: {e}")
        finally:
            cursor.close()
            connection.close()

if __name__ == "__main__":
    create_database_and_table()

# Example usage
# create_sale('Product A', 10, '2023-12-09 10:00:00', 'Branch A')
# read_sales()
# update_sale(1, 'Product B', 20, '2023-12-10 11:00:00', 'Branch B')
# delete_sale(1)
```

Other Table Operations - Drop Table, Limit, Joining Tables

```
def drop_table(table_name):
    connection = connect()
    if connection:
        try:
            cursor = connection.cursor()
            drop_query = f"DROP TABLE IF EXISTS {table_name}"
            cursor.execute(drop_query)
            connection.commit()
            print(f"Table {table_name} dropped successfully")
        except Error as e:
            print(f"Error: {e}")
        finally:
            cursor.close()
            connection.close()

def limit_sales(limit_number):
    connection = connect()
    if connection:
        try:
            cursor = connection.cursor()
            cursor.execute(f"SELECT * FROM sales LIMIT {limit_number}")
            rows = cursor.fetchall()
            for row in rows:
                print(row)
        except Error as e:
            print(f"Error: {e}")
        finally:
            cursor.close()
            connection.close()

def join_tables(table1, table2, join_type, on_condition):
    connection = connect()
    if connection:
        try:
            cursor = connection.cursor()
            join_query = f"SELECT * FROM {table1} {join_type} JOIN {table2} ON {on_condition}"
            cursor.execute(join_query)
            rows = cursor.fetchall()
            for row in rows:
                print(row)
        except Error as e:
            print(f"Error: {e}")
        finally:
            cursor.close()
            connection.close()
```

Removing List Duplicates

Using a Set (Order Not Preserved):
A set is a collection type that does not allow duplicates. However, converting a list to a set does not preserve the original order of the elements.

```
my_list = [1, 2, 2, 3, 4, 4, 5]
no_duplicates = list(set(my_list))
```

Using a Set with Order Preservation (Python 3.6+):
By iterating over the list and adding each element to a set for quick membership tests, you can preserve the order while removing duplicates. In Python 3.6 and above, dictionaries and consequently sets retain insertion order.

```
my_list = [1, 2, 2, 3, 4, 4, 5]
seen = set()
no_duplicates = []
for item in my_list:
    if item not in seen:
        seen.add(item)
        no_duplicates.append(item)
```

Using List Comprehension (Order Preserved):
This is a more concise version of the previous method, using list comprehension.

```
my_list = [1, 2, 2, 3, 4, 4, 5]
seen = set()
no_duplicates = [x for x in my_list if not (x in seen or seen.add(x))]
```

Using a Dictionary (Order Preserved, Python 3.7+):
Similar to using a set, but this method utilizes the fact that dictionaries in Python 3.7 and later versions are ordered.

```
my_list = [1, 2, 2, 3, 4, 4, 5]
no_duplicates = list(dict.fromkeys(my_list))
```

Simple UI for the Python MySQL CRUD Operations

Example 1:

```
import tkinter as tk
from tkinter import messagebox, simpledialog
import pymysql
from pymysql import Error

# Database connection function
def connect_to_db():
    try:
        return pymysql.connect(
            host='localhost',
            user='john',
            password='123',
            database='ppsta'
        )
    except Error as e:
        messagebox.showerror("Connection Error", f"Error connecting to database: {e}")
        return None

# Function to create database and table
def create_db_and_table():
    connection = connect_to_db()
    if connection:
        try:
            cursor = connection.cursor()
            cursor.execute("CREATE DATABASE IF NOT EXISTS ppsta")
            cursor.execute("USE ppsta")
            create_table_query = """
            CREATE TABLE IF NOT EXISTS sales (
                salesid INT AUTO_INCREMENT PRIMARY KEY,
```

```
            productname VARCHAR(255),
            quantity INT,
            transactiondate DATETIME,
            branch VARCHAR(255)
        )
        """
        cursor.execute(create_table_query)
        connection.commit()
        messagebox.showinfo("Success", "Database and table created successfully")
    except Error as e:
        messagebox.showerror("Error", f"Error: {e}")
    finally:
        cursor.close()
        connection.close()

# CRUD operations
def create_sale():
    connection = connect_to_db()
    if connection:
        try:
            cursor = connection.cursor()
            insert_query = """
            INSERT INTO sales (productname, quantity, transactiondate, branch)
            VALUES (%s, %s, %s, %s)
            """
            cursor.execute(insert_query, (productname_var.get(), quantity_var.get(), transactiondate_var.get(),
branch_var.get()))
            connection.commit()
            messagebox.showinfo("Success", "Sale record created successfully")
        except Error as e:
            messagebox.showerror("Error", f"Error: {e}")
        finally:
            cursor.close()
            connection.close()

def read_sales():
    connection = connect_to_db()
    if connection:
        try:
            cursor = connection.cursor()
            cursor.execute("SELECT * FROM sales")
            rows = cursor.fetchall()
            result = "\n".join([str(row) for row in rows])
            messagebox.showinfo("Sales Records", result)
        except Error as e:
            messagebox.showerror("Error", f"Error: {e}")
        finally:
            cursor.close()
            connection.close()

def update_sale():
    salesid = simpledialog.askinteger("Input", "Enter salesid to update")
    if salesid is None:
        return
    connection = connect_to_db()
    if connection:
        try:
            cursor = connection.cursor()
```

```
            update_query = """
            UPDATE sales
            SET productname = %s, quantity = %s, transactiondate = %s, branch = %s
            WHERE salesid = %s
            """
            cursor.execute(update_query, (productname_var.get(), quantity_var.get(), transactiondate_var.get(),
branch_var.get(), salesid))
            connection.commit()
            messagebox.showinfo("Success", "Sale record updated successfully")
        except Error as e:
            messagebox.showerror("Error", f"Error: {e}")
        finally:
            cursor.close()
            connection.close()

def delete_sale():
    salesid = simpledialog.askinteger("Input", "Enter salesid to delete")
    if salesid is None:
        return
    connection = connect_to_db()
    if connection:
        try:
            cursor = connection.cursor()
            delete_query = "DELETE FROM sales WHERE salesid = %s"
            cursor.execute(delete_query, (salesid,))
            connection.commit()
            messagebox.showinfo("Success", "Sale record deleted successfully")
        except Error as e:
            messagebox.showerror("Error", f"Error: {e}")
        finally:
            cursor.close()
            connection.close()

# Tkinter window setup
root = tk.Tk()
root.title("Database Management")

# Entry variables
productname_var = tk.StringVar()
quantity_var = tk.StringVar()
transactiondate_var = tk.StringVar()
branch_var = tk.StringVar()

# UI for CRUD operations
tk.Label(root, text="Product Name").pack()
tk.Entry(root, textvariable=productname_var).pack()

tk.Label(root, text="Quantity").pack()
tk.Entry(root, textvariable=quantity_var).pack()

tk.Label(root, text="Transaction Date (YYYY-MM-DD HH:MM:SS)").pack()
tk.Entry(root, textvariable=transactiondate_var).pack()

tk.Label(root, text="Branch").pack()
tk.Entry(root, textvariable=branch_var).pack()

# Create database and table button
create_db_btn = tk.Button(root, text="Create Database and Table", command=create_db_and_table)
```

```
create_db_btn.pack(pady=10)

# CRUD operation buttons
tk.Button(root, text="Create Sale Record", command=create_sale).pack(pady=5)
tk.Button(root, text="Read Sale Records", command=read_sales).pack(pady=5)
tk.Button(root, text="Update Sale Record", command=update_sale).pack(pady=5)
tk.Button(root, text="Delete Sale Record", command=delete_sale).pack(pady=5)

root.mainloop()
```

Example 2: Display records in a table

```
import tkinter as tk
from tkinter import ttk, messagebox, simpledialog
import pymysql
from pymysql import Error

# ... (Your existing functions: connect_to_db, create_db_and_table, etc.)
# Database connection function
def connect_to_db():
    try:
        return pymysql.connect(
            host='localhost',
            user='john',
            password='123',
            database='ppsta'
        )
    except Error as e:
        messagebox.showerror("Connection Error", f"Error connecting to database: {e}")
        return None

# Function to create database and table
def create_db_and_table():
    connection = connect_to_db()
    if connection:
        try:
            cursor = connection.cursor()
            cursor.execute("CREATE DATABASE IF NOT EXISTS ppsta")
            cursor.execute("USE ppsta")
            create_table_query = """
            CREATE TABLE IF NOT EXISTS sales (
                salesid INT AUTO_INCREMENT PRIMARY KEY,
                productname VARCHAR(255),
                quantity INT,
                transactiondate DATETIME,
                branch VARCHAR(255)
            )
            """
            cursor.execute(create_table_query)
            connection.commit()
            messagebox.showinfo("Success", "Database and table created successfully")
        except Error as e:
            messagebox.showerror("Error", f"Error: {e}")
        finally:
            cursor.close()
            connection.close()
```

```
# Function to refresh the sales records in the Treeview
def refresh_sales_records():
    for record in tree.get_children():
        tree.delete(record)
    connection = connect_to_db()
    if connection:
        try:
            cursor = connection.cursor()
            cursor.execute("SELECT * FROM sales")
            rows = cursor.fetchall()
            for row in rows:
                tree.insert("", tk.END, values=row)
        except Error as e:
            messagebox.showerror("Error", f"Error: {e}")
        finally:
            cursor.close()
            connection.close()

# Modified CRUD operations to include refresh_sales_records()
def create_sale():
    # ... (Your existing create_sale function)
    connection = connect_to_db()
    if connection:
        try:
            cursor = connection.cursor()
            insert_query = """
            INSERT INTO sales (productname, quantity, transactiondate, branch)
            VALUES (%s, %s, %s, %s)
            """
            cursor.execute(insert_query, (productname_var.get(), quantity_var.get(), transactiondate_var.get(),
branch_var.get()))
            connection.commit()
            messagebox.showinfo("Success", "Sale record created successfully")
        except Error as e:
            messagebox.showerror("Error", f"Error: {e}")
        finally:
            cursor.close()
            connection.close()

    refresh_sales_records()

def read_sales():
    connection = connect_to_db()
    if connection:
        try:
            cursor = connection.cursor()
            cursor.execute("SELECT * FROM sales")
            rows = cursor.fetchall()
            result = "\n".join([str(row) for row in rows])
            messagebox.showinfo("Sales Records", result)
        except Error as e:
            messagebox.showerror("Error", f"Error: {e}")
        finally:
            cursor.close()
            connection.close()

    refresh_sales_records()
```

```
def update_sale():
    # ... (Your existing update_sale function)
    salesid = simpledialog.askinteger("Input", "Enter salesid to update")
    if salesid is None:
        return
    connection = connect_to_db()
    if connection:
        try:
            cursor = connection.cursor()
            update_query = """
            UPDATE sales
            SET productname = %s, quantity = %s, transactiondate = %s, branch = %s
            WHERE salesid = %s
            """
            cursor.execute(update_query, (productname_var.get(), quantity_var.get(), transactiondate_var.get(),
branch_var.get(), salesid))
            connection.commit()
            messagebox.showinfo("Success", "Sale record updated successfully")
        except Error as e:
            messagebox.showerror("Error", f"Error: {e}")
        finally:
            cursor.close()
            connection.close()

    refresh_sales_records()

def delete_sale():
    # ... (Your existing delete_sale function)
    salesid = simpledialog.askinteger("Input", "Enter salesid to delete")
    if salesid is None:
        return
    connection = connect_to_db()
    if connection:
        try:
            cursor = connection.cursor()
            delete_query = "DELETE FROM sales WHERE salesid = %s"
            cursor.execute(delete_query, (salesid,))
            connection.commit()
            messagebox.showinfo("Success", "Sale record deleted successfully")
        except Error as e:
            messagebox.showerror("Error", f"Error: {e}")
        finally:
            cursor.close()
            connection.close()


    refresh_sales_records()

# Tkinter window setup
root = tk.Tk()
root.title("Database Management")

# ... (Your existing UI setup for CRUD operations)
# Entry variables
productname_var = tk.StringVar()
quantity_var = tk.StringVar()
transactiondate_var = tk.StringVar()
branch_var = tk.StringVar()
```

```
# UI for CRUD operations
tk.Label(root, text="Product Name").pack()
tk.Entry(root, textvariable=productname_var).pack()

tk.Label(root, text="Quantity").pack()
tk.Entry(root, textvariable=quantity_var).pack()

tk.Label(root, text="Transaction Date (YYYY-MM-DD HH:MM:SS)").pack()
tk.Entry(root, textvariable=transactiondate_var).pack()

tk.Label(root, text="Branch").pack()
tk.Entry(root, textvariable=branch_var).pack()

# Create database and table button
create_db_btn = tk.Button(root, text="Create Database and Table", command=create_db_and_table)
create_db_btn.pack(pady=10)

# CRUD operation buttons
tk.Button(root, text="Create Sale Record", command=create_sale).pack(pady=5)
tk.Button(root, text="Read Sale Records", command=read_sales).pack(pady=5)
tk.Button(root, text="Update Sale Record", command=update_sale).pack(pady=5)
tk.Button(root, text="Delete Sale Record", command=delete_sale).pack(pady=5)

# Treeview (Table) setup
columns = ("salesid", "productname", "quantity", "transactiondate", "branch")
tree = ttk.Treeview(root, columns=columns, show='headings')
for col in columns:
    tree.heading(col, text=col.capitalize())
tree.column("salesid", width=50)  # Adjust column width as needed
tree.column("productname", width=100)
tree.column("quantity", width=50)
tree.column("transactiondate", width=150)
tree.column("branch", width=100)

# Adding a scrollbar
scrollbar = ttk.Scrollbar(root, orient=tk.VERTICAL, command=tree.yview)
tree.configure(yscroll=scrollbar.set)
scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
tree.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

# Initial refresh of data
refresh_sales_records()

root.mainloop()
```

Data Visualization with Python and MySQL

Using Matplotlib (Basic Plotting)

*Note: Using PyMySQL vs SQLALCHEMY

```
import pymysql
import pandas as pd

# Database connection parameters
config = {
    'user': 'john',
    'password': '123',
    'host': 'localhost',
```

```
    'database': 'ppsta',  # Replace with your database name
}

# Connect to the database
try:
    # Establish a database connection using PyMySQL
    cnx = pymysql.connect(**config)
    print("Successfully connected to the database.")

    # Load 'sales' table into a DataFrame
    query = "SELECT * FROM sales;"
    sales_df = pd.read_sql(query, cnx)

    # Close the database connection
    cnx.close()

except pymysql.Error as err:
    print(f"Error: {err}")

sales_df
```

*The code above works normally outside Jupyter Notebooks but throws error when used in notebooks:

```
C:\Users\core360\AppData\Local\Temp\ipykernel_9572\3900321011.py:20: UserWarning: pandas only supports SQLAlchemy connectable (engine/connection) or database string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested. Please consider using SQLAlchemy.
  sales_df = pd.read_sql(query, cnx)
```

*Let us use SQLAlchemy

```
from sqlalchemy import create_engine
import pandas as pd

# Database connection parameters
username = 'john'
password = '123'
host = 'localhost'
database = 'ppsta'  # Replace with your database name

# Create a connection string and engine
connection_string = f"mysql+pymysql://{username}:{password}@{host}/{database}"
engine = create_engine(connection_string)

try:
    # Load 'sales' table into a DataFrame
    sales_df = pd.read_sql("sales", engine)

except Exception as e:
    print(f"An error occurred: {e}")

sales_df
```

Logic, Control Flow and Filtering

To select specific fields (or columns) from a pandas DataFrame, you can use simple indexing syntax. Assuming you have a DataFrame named df and you want to select columns named 'Column1' and 'Column2', you would do it as follows:

```
selected_columns = df[['Column1', 'Column2']]
```

To add conditional filtering to a pandas DataFrame, you can use boolean indexing. This allows you to filter the rows of the DataFrame based on the values of certain columns.

```
filtered_df = df[(df['Column1'] > 5) & (df['Column3'] < 8)]
print(filtered_df)
```

Notes

- Use & for logical AND, | for logical OR, and ~ for logical NOT when combining conditions.
- Always wrap individual conditions in parentheses due to Python's operator precedence rules.
- You can create as complex a condition as necessary by combining these logical operators.

Here are more examples of logic, control flow, and filtering techniques in pandas DataFrames. These examples demonstrate various common scenarios you might encounter while working with pandas.

Example 1: Using .isin() for Filtering

Suppose you have a DataFrame and want to filter rows based on whether the values in a certain column match any value in a given list.

```
df = pd.DataFrame({
    'Color': ['Red', 'Blue', 'Green', 'Yellow', 'White', 'Black'],
    'Value': [5, 10, 15, 20, 25, 30]
})
```

```
# Filter rows where 'Color' is either 'Red' or 'Black'
filtered_df = df[df['Color'].isin(['Red', 'Black'])]
print(filtered_df)
```

Example 2: Using .query() for Complex Filtering

The .query() method allows for filtering using a query string, which can be more readable, especially for complex conditions.

```
# Filter rows where 'Value' is greater than 10 and 'Color' is not 'White'
filtered_df = df.query("Value > 10 and Color != 'White'")
print(filtered_df)
```

Example 3: Conditional Column Creation

You can create new columns in a DataFrame based on conditional logic.

```
# Create a new column 'Category' based on the value in 'Value'
df['Category'] = df['Value'].apply(lambda x: 'High' if x > 15 else 'Low')
print(df)
```

Example 4: Using np.where() for Conditional Assignments

numpy.where() is useful for creating a new column based on conditions.

```
import numpy as np

# Create a new column 'Size' based on 'Value': 'Large' if 'Value' > 20, else 'Small'
df['Size'] = np.where(df['Value'] > 20, 'Large', 'Small')
print(df)
```

Example 5: Combining Multiple Conditions

You can combine multiple conditions using & (AND) and | (OR) operators.

```
# Filter rows where 'Value' is greater than 10 but less than 30
filtered_df = df[(df['Value'] > 10) & (df['Value'] < 30)]
print(filtered_df)
```

Example 6: Filtering with String Methods

Pandas string methods under .str can be used to perform vectorized string operations, useful for filtering based on string conditions.

```
# Filter rows where 'Color' starts with the letter 'B'
filtered_df = df[df['Color'].str.startswith('B')]
print(filtered_df)
```

Example 7: Filtering Based on Index

You can also filter rows based on the index values.

```
# Setting 'Color' as the index
df.set_index('Color', inplace=True)
```

```
# Filter rows where the index is either 'Red' or 'Green'
filtered_df = df.loc[['Red', 'Green']]
print(filtered_df)
```

Matplotlib Demo Manipulating Graphs Properties (Font, Size, Color Scheme)

Matplotlib is a powerful Python library for creating static, interactive, and animated visualizations. When combined with pandas, it becomes very convenient to plot data directly from DataFrames.

1. Import Libraries

```
import pandas as pd
import matplotlib.pyplot as plt
```

2. Create a Sample DataFrame

```
data = {
    'Year': [2015, 2016, 2017, 2018, 2019],
    'Sales': [200, 300, 400, 500, 600],
    'Expenses': [150, 200, 250, 300, 350]
}
df = pd.DataFrame(data)
```

3. Basic Plotting

Line Plot

Line plots are useful for showing trends over time. You can create a line plot directly from a DataFrame:

```
# Plotting
df.plot(x='Year', y='Sales', kind='line')
plt.title('Sales Over Years')
plt.ylabel('Sales')
plt.xlabel('Year')
plt.show()
```

Bar Chart

Bar charts are great for comparing different groups or categories:

```
# Bar chart
df.plot(x='Year', y='Sales', kind='bar')
plt.title('Sales Per Year')
plt.ylabel('Sales')
plt.xlabel('Year')
plt.show()
```

Multiple Lines

To plot multiple lines on the same graph:

```
df.plot(x='Year', y=['Sales', 'Expenses'], kind='line')
plt.title('Sales and Expenses Over Years')
```

```
plt.ylabel('Value')
plt.xlabel('Year')
plt.show()
```

4. Customizing Plots

Matplotlib offers extensive options for customizing plots:

- Changing Colors and Styles: You can change line colors, types, marker styles, etc.
- Adding Grids, Labels, Titles: Improve readability by adding grids, labels, and titles to your plot.
- Multiple Plots: You can create subplots to show multiple plots in the same figure.

Example: Customized Line Plot

```
df.plot(x='Year', y='Sales', kind='line', color='red', linestyle='--', marker='o')
plt.title('Sales Over Years')
plt.ylabel('Sales')
plt.xlabel('Year')
plt.grid(True)
plt.show()
```

5. Saving Plots

You can save your plots to files:

```
df.plot(x='Year', y='Sales', kind='line')
plt.title('Sales Over Years')
plt.ylabel('Sales')
plt.xlabel('Year')
plt.savefig('sales_over_years.png')
```

Module 3: Setting up Data Science and Analytics Visualization Environment

Installing and configuring Jupyter Notebook

1. Install python 3.x.x, set global
2. *Upgrade pip

```
\> python -m pip install --upgrade pip
```

3. Install jupyter notebook

```
\> pip install notebook
```

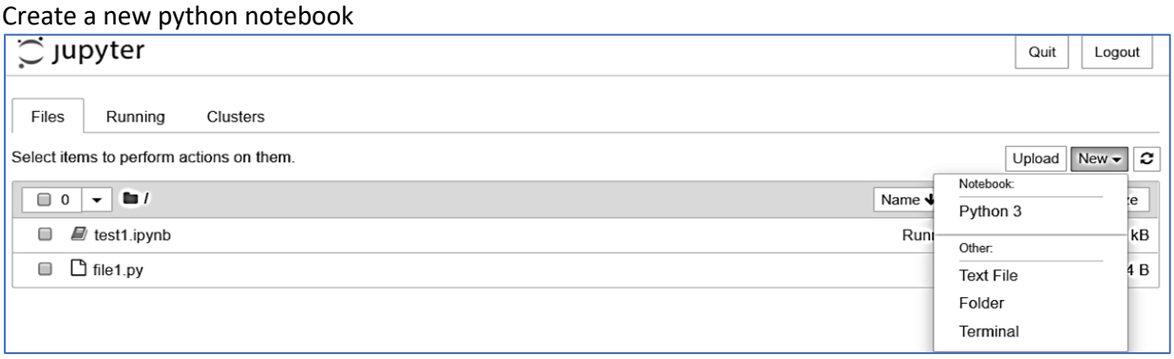
4. Create a directory for your notebook
5. Enter directory and setup configuration file and create password

```
\> jupyter notebook –generate-config
\> jupyter notebook password
```

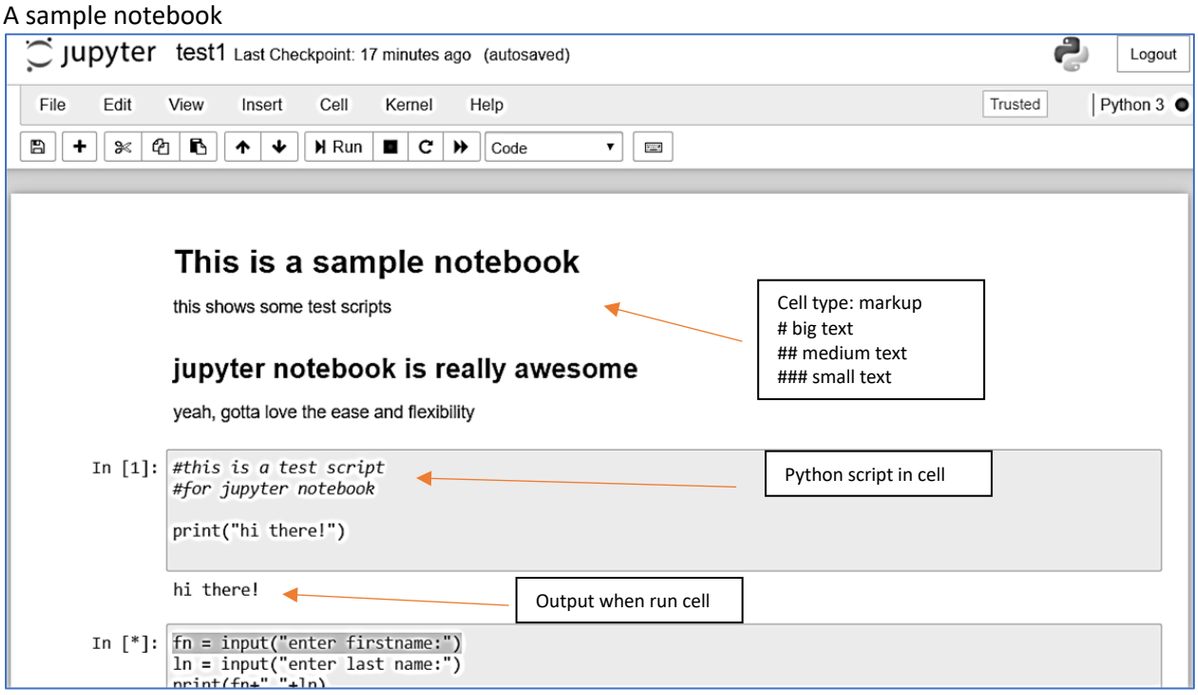
6. Enter directory and launch jupyter server

```
\> jupyter notebook
\> jupyter notebook --ip <your ip> --port 8888
```

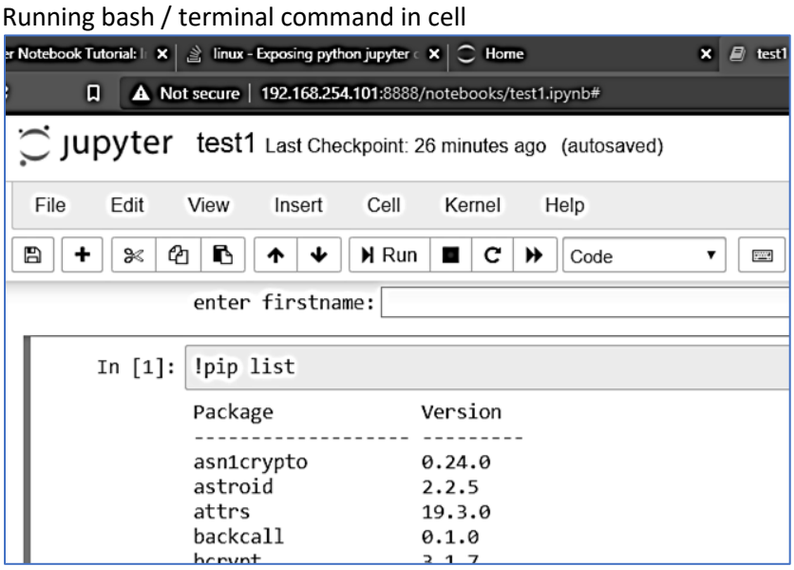
7. Create a new python notebook



8. A sample notebook



9. Running bash / terminal command in cell



10. To export and download your notebook as pdf

```
\> pip install pandoc
\> pip install nbconvert
\> pip install nbconvert[webpdf]
\> playwright install chromium
```

Then in your notebook, File→Save and Export Notebook As…→webpdf

Hiding Code Cells in Jupyter Notebook
View → Collapse all Code

Module 4: Mathematical Computing with Python (NumPy)

NumPy aims to provide an array object up to 50x faster than traditional python lists. It provides a lot of supporting functions working with NumPy arrays easier.

Import the package

```
import numpy as np
```

creating a 1D Array

```
ar=np.array([1,4,6,3,7])
ar
[output] array([1, 4, 6, 3, 7])

ar1=np.array((2,3,4,5))
ar1
[output] array([2, 3, 4, 5])
```

create a 2-D Array

```
arr=np.array([[1,2,4], [5,6,8]])
arr
[output] array([[1, 2, 4], [5, 6, 8]])
```

create a 3-D Array

```
arr=np.array([[[1,2,3], [5,6,7]], [[1,2,3], [5,6,7]]])
arr
[output] array([[[1, 2, 3],
                [5, 6, 7]],
                [[1, 2, 3],
                [5, 6, 7]]])
```

#accessing elements

```
ar=np.array([1,4,6,3,7])
ar[2]
[output] 6

arr=np.array([[1,2,4], [5,6,8]])
arr[0,1]
[output] 2
```

Use negative indexing to access an array from the end.

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('Last element from 2nd dim: ', arr[1, -1])
```

Mathematical Functions of NumPy

We can perform

- ✓ Trigonometric Functions
- ✓ Logarithmic Functions
- ✓ Inverse Functions
- ✓ Exponential Functions
- ✓ Statistical Functions

Unary operations

get cosine values

```
arr=np.array([1,3,5,2,6])
np.cos(arr)
[output] array([ 0.54030231, -0.9899925 ,  0.28366219, -0.41614684,  0.96017029])
```

get logarithmic values

```
np.log(arr)
[output] array([0.        , 1.09861229, 1.60943791, 0.69314718, 1.79175947])
```

get the mean, median and standard deviations

```
np.mean(arr)
[output] 3.4
```

```
np.median(arr)
[output] 3.0
```

```
np.std(arr)
1.8547236990991407
```

return exponential values in form of an array

```
np.exp(arr)
array([  2.71828183,  20.08553692, 148.4131591 ,   7.3890561 ,      403.42879349])
```

Binary Operations

create sample arrays

```
arr1=np.array([2,4,6,8,2])
arr2=np.array([3,6,1,5,2])
```

#addition

```
arr1+arr2
[output] array([ 5, 10,  7, 13,  4])
```

#subtraction

```
arr1-arr2
[output] array([ 5, 10,  7, 13,  4])
```

#multiplication

```
arr1*arr2
[output] array([ 6, 24,  6, 40,  4])
```

More NumPy Built-In Functions

import package

```
import numpy as np
```

create a 1D array that are all 1s

```
arr=np.ones(2)
arr
[output] array([1., 1.])
```

create 2D array with all values 1s

```
np.ones((2,2))
[output] array([[1., 1.],   [1., 1.]])
```

create a 1D array that are all 0s

```
np.zeros(2)
{output} array([0., 0.])
```

create 2D array with all values 0s

```
np.zeros((2,2))
[output] array([[0., 0.],     [0., 0.]])
```

returns an integer that tells us how many dimensions the array have.

```
import numpy as np

a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

An array can have any number of dimensions.
When the array is created, you can define the number of dimensions by using the ndmin argument.

```
import numpy as np

arr = np.array([1, 2, 3, 4], ndmin=5)

print(arr)
print('number of dimensions :', arr.ndim)
```

Slicing Arrays

- ✓ Slicing in python means taking elements from one given index to another given index.
- ✓ We pass slice instead of index like this: [start:end].
- ✓ We can also define the step, like this: [start:end:step].
- ✓ If we don't pass start its considered 0
- ✓ If we don't pass end its considered length of array in that dimension
- ✓ If we don't pass step its considered 1

Slice elements from index 1 to index 5 from the following array:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5])
```

Slice elements from index 4 to the end of the array:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[4:])
```

Slice elements from the beginning to index 4 (not included):

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[:4])
```

Use the minus operator to refer to an index from the end:
Slice from the index 3 from the end to index 1 from the end:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[-3:-1])
```

Use the step value to determine the step of the slicing:
Return every other element from index 1 to index 5:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5:2])
```

Return every other element from the entire array:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[::2])
```

Slicing 2-D Arrays
Note: Remember that second element has index 1.
From the second element, slice elements from index 1 to index 4 (not included):

```
import numpy as np
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[1, 1:4])
```

NumPy Array Copy vs View

The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.

The copy owns the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy.

The view does not own the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view.

Make a copy, change the original array, and display both arrays:
The copy SHOULD NOT be affected by the changes made to the original array.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
arr[0] = 42
print(arr)
print(x)
```

Make a view, change the original array, and display both arrays:
The view SHOULD be affected by the changes made to the original array.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
arr[0] = 42
print(arr)
print(x)
```

Make a view, change the view, and display both arrays:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
x[0] = 31
print(arr)
print(x)
```

Print the value of the base attribute to check if an array owns it's data or not:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
y = arr.view()
print(x.base)
print(y.base)
```

Shape of an Array

The shape of an array is the number of elements in each dimension.

NumPy arrays have an attribute called shape

Shape returns a tuple with each index having the number of corresponding elements.

Print the shape of a 2-D array:

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(arr.shape)

# The example above returns (2, 4), which means that the array has 2 dimensions, where the first dimension has 2 elements and the second has 4.
```

Create an array with 5 dimensions using ndmin using a vector with values 1,2,3,4 and verify that last dimension has value 4:

```
import numpy as np
arr = np.array([1, 2, 3, 4], ndmin=5)
print(arr)
print('shape of array :', arr.shape)
```

Reshaping arrays

By reshaping we can add or remove dimensions or change number of elements in each dimension.

Convert the following 1-D array with 12 elements into a 2-D array.

The outermost dimension will have 4 arrays, each with 3 elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(4, 3)
print(newarr)
```

Convert the following 1-D array with 12 elements into a 3-D array.

The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(2, 3, 2)
print(newarr)
```

Iterating

Iterate on the elements of the following 1-D array:

```
import numpy as np
arr = np.array([1, 2, 3])
for x in arr:
  print(x)
```

Iterate on the elements of the following 2-D array:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
for x in arr:
  print(x)
```

Iterate on each scalar element of the 2-D array:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
for x in arr:
  for y in x:
    print(y)
```

Iterate on the elements of the following 3-D array:

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
for x in arr:
  print(x)
```

Iterate down to the scalars:

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
for x in arr:
  for y in x:
    for z in y:
      print(z)
```

Iterating Arrays Using nditer()

In basic for loops, iterating through each scalar of an array we need to use n for loops which can be difficult to write for arrays with very high dimensionality.

Iterate through the following 3-D array:

```
import numpy as np
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
for x in np.nditer(arr):
  print(x)
```

Enumerate on following 1D arrays elements:

```
import numpy as np
arr = np.array([1, 2, 3])
for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

Enumerate on following 2D array's elements:

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

Join

Join two arrays

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.concatenate((arr1, arr2))
print(arr)
```

Join two 2-D arrays along rows (axis=1):

```
import numpy as np
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])
arr = np.concatenate((arr1, arr2), axis=1)
print(arr)
```

We can concatenate two 1-D arrays along the second axis
this would result in putting them one over the other, ie. stacking.

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
```

```
arr = np.stack((arr1, arr2), axis=1)
print(arr)
```

NumPy provides a helper function: hstack() to stack along rows.

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.hstack((arr1, arr2))
print(arr)
```

NumPy provides a helper function: vstack() to stack along columns.

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.vstack((arr1, arr2))
print(arr)
```

Splitting

Split the array in 3 parts:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
newarr = np.array_split(arr, 3)
print(newarr)
```

Access the splitted arrays:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
newarr = np.array_split(arr, 3)
print(newarr[0])
print(newarr[1])
print(newarr[2])
```

Split the 2-D array into three 2-D arrays.

```
import numpy as np
arr = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])
newarr = np.array_split(arr, 3)
print(newarr)
```

Searching

Find the indexes where the value is 4:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 4, 4])
x = np.where(arr == 4)
print(x)
```

Find the indexes where the values are even:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
x = np.where(arr%2 == 0)
print(x)
```

Sorting

Sort the array:

```
import numpy as np
arr = np.array([3, 2, 0, 1])
print(np.sort(arr))
```

Sort the array alphabetically:

```
import numpy as np
arr = np.array(['banana', 'cherry', 'apple'])
print(np.sort(arr))
```

Sort a 2-D array:

```
import numpy as np
arr = np.array([[3, 2, 4], [5, 0, 1]])
print(np.sort(arr))
```

Data Distribution

Data Distribution is a list of all possible values, and how often each value occurs. The random module offer methods that returns randomly generated data distributions.

- We can generate random numbers based on defined probabilities using the choice() method of the random module.
- The choice() method allows us to specify the probability for each value.
- he probability is set by a number between 0 and 1, where 0 means that the value will never occur and 1 means that the value will always occur.

#Generate a 1-D array containing 100 values, where each value has to be 3, 5, 7 or 9.
The probability for the value to be 3 is set to be 0.1
The probability for the value to be 5 is set to be 0.3
The probability for the value to be 7 is set to be 0.6
The probability for the value to be 9 is set to be 0

```
from numpy import random
x = random.choice([3, 5, 7, 9], p=[0.1, 0.3, 0.6, 0.0], size=(100))
print(x)
```

- The sum of all probability numbers should be 1.
- Even if you run the example above 100 times, the value 9 will never occur.
- You can return arrays of any shape and size by specifying the shape in the size parameter.

Same example as above, but return a 2-D array with 3 rows, each containing 5 values.

```
from numpy import random
x = random.choice([3, 5, 7, 9], p=[0.1, 0.3, 0.6, 0.0], size=(3, 5))
print(x)
```

Module 5: Using Padas Visualization and Plotly Interactive Plots
Introduction to Pandas

Pandas is a high-level data manipulation tool used in machine learning and data science.

To install pandas (comes installed with Jupyter)

```
\> pip install pandas
```

To use, import in your script

```
import pandas as pd
```

Reading Datasets

By calling read_csv(), read_excel() and other datasets, you create a DataFrame, which is the main data structure used in pandas.

Reading CSV files

```
data=pd.read_csv('Iris.csv')
data.head()
```

Reading Excel File

```
data1=pd.read_excel('example_excel.xls')
data1.head()
```

Plotting Datasets using Pandas

import package

```
import pandas as pd
```

read and test-display some data

```
data=pd.read_csv('Iris.csv')
data.head()
```

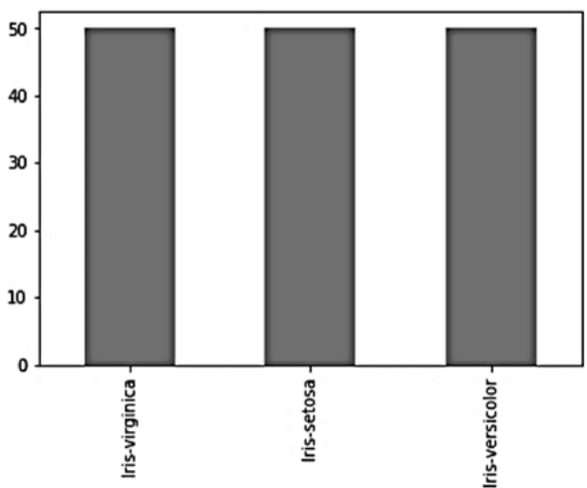
Note:

- ✓ The value_counts() function is used to return the value for different classes present in the column of a dataset.
- ✓ Plot a Pie Chart when we have 4-5 distinct values in a column, a Bar Chart when we have more than 5 distinct values and a line chart if we have many distinct values

create a bar plot

```
data['Species'].value_counts().plot(kind='bar')
```

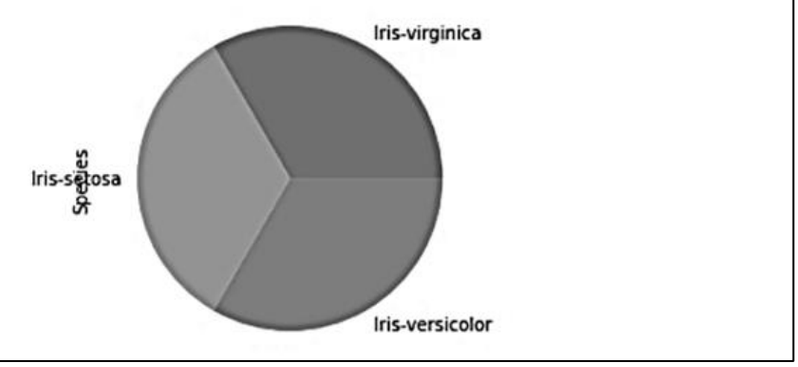
```
Out[17]: <matplotlib.axes._subplots.AxesSubplot at 0x18e266802c8>
```



create a pie chart

```
data['Species'].value_counts().plot(kind='pie')
```

```
Out[18]: <matplotlib.axes._subplots.AxesSubplot at 0x18e266ed408>
```



create a line chart

```
data['SepalLengthCm'].value_counts().plot(kind='line')
```

Understanding DataFrame

Activity:

First, download the data by passing the download URL to pandas.read_csv():

```
import pandas as pd
download_url = (
"https://raw.githubusercontent.com/fivethirtyeight/"
"data/master/college-majors/recent-grads.csv"
)
```

```
df = pd.read_csv(download_url)
```

```
type(df)
Out[4]: pandas.core.frame.DataFrame
```

Now that you have a DataFrame, you can take a look at the data.
First, you should configure the display.max.columns option to make sure pandas doesn't hide any columns.
Then you can view the first few rows of data with .head():

```
pd.set_option("display.max.columns", None)
df.head()
```

Your dataset contains some columns related to the earnings of graduates in each major:

- "Median" is the median earnings of full-time, year-round workers.
- "P25th" is the 25th percentile of earnings.
- "P75th" is the 75th percentile of earnings.
- "Rank" is the major's rank by median earnings.

Now you're ready to make your first plot with .plot():

```
df.plot(x="Rank", y=["P25th", "Median", "P75th"])
```

note: .plot() returns a line graph containing data from every row in the DataFrame.
The x-axis values represent the rank of each institution
The "P25th", "Median", and "P75th" values are plotted on the y-axis.

Looking at the plot, you can make the following observations:

- ✓ The median income decreases as rank decreases. This is expected because the rank is determined by the median income.
- ✓ Some majors have large gaps between the 25th and 75th percentiles. People with these degrees may earn significantly less or significantly more than the median income.
- ✓ Other majors have very small gaps between the 25th and 75th percentiles. People with these degrees earn salaries very close to the median income.

Plot Types

.plot() has several optional parameters. Most notably, the kind parameter accepts eleven different string values and determines which kind of plot you'll create:

- ✓ "area" is for area plots.
- ✓ "bar" is for vertical bar charts.
- ✓ "barh" is for horizontal bar charts.
- ✓ "box" is for box plots.
- ✓ "hexbin" is for hexbin plots.
- ✓ "hist" is for histograms.
- ✓ "kde" is for kernel density estimate charts.
- ✓ "density" is an alias for "kde".
- ✓ "line" is for line graphs.

- ✓ "pie" is for pie charts.
- ✓ "scatter" is for scatter plots.

The default value is "line". Line graphs, like the one you created above, provide a good overview of your data. You can use them to detect general trends. They rarely provide sophisticated insight, but they can give you clues as to where to zoom in.

If you don't provide a parameter to .plot(), then it creates a line plot with the index on the x-axis and all the numeric columns on the y-axis. While this is a useful default for datasets with only a few columns, for the college majors dataset and its several numeric columns, it looks like quite a mess.

Distributions and Histograms

DataFrame is not the only class in pandas with a .plot() method. As so often happens in pandas, the Series object provides similar functionality.

You can get each column of a DataFrame as a Series object. Here's an example using the "Median" column of the DataFrame you created from the college major data:

```
median_column = df["Median"]
```

```
type(median_column)
Out[13]: pandas.core.series.Series
```

```
median_column.plot(kind="hist")
```

The histogram shows the data grouped into ten bins ranging from $20,000 to $120,000, and each bin has a width of $10,000. The histogram has a different shape than the normal distribution, which has a symmetric bell shape with a peak in the middle. The histogram of the median data, however, peaks on the left below $40,000. The tail stretches far to the right and suggests that there are indeed fields whose majors can expect significantly higher earnings.

Sorting for plotting

Contrary to the first overview, you only want to compare a few data points, but you want to see more details about them. For this, a bar plot is an excellent tool.

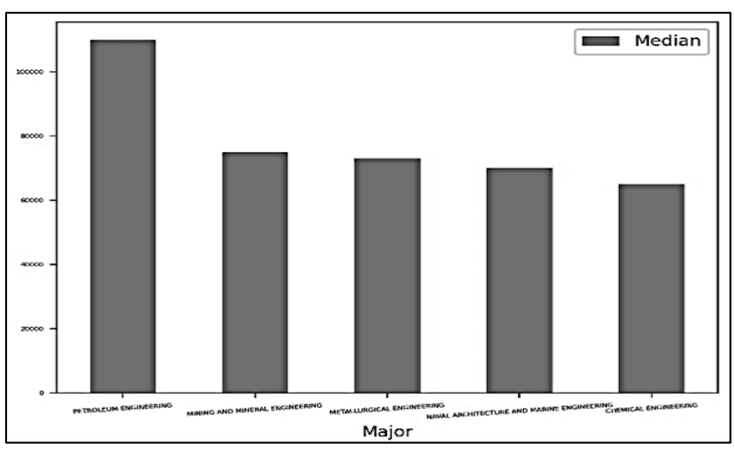
select the five majors with the highest median earnings.

```
top_5 = df.sort_values(by="Median", ascending=False).head()
```

create a bar plot that shows only the majors with these top five median salaries:

Notice that you use the rot and fontsize parameters to rotate and size the labels of the x-axis so that they're visible.

```
top_5.plot(x="Major", y="Median", kind="bar", rot=5, fontsize=4)
```



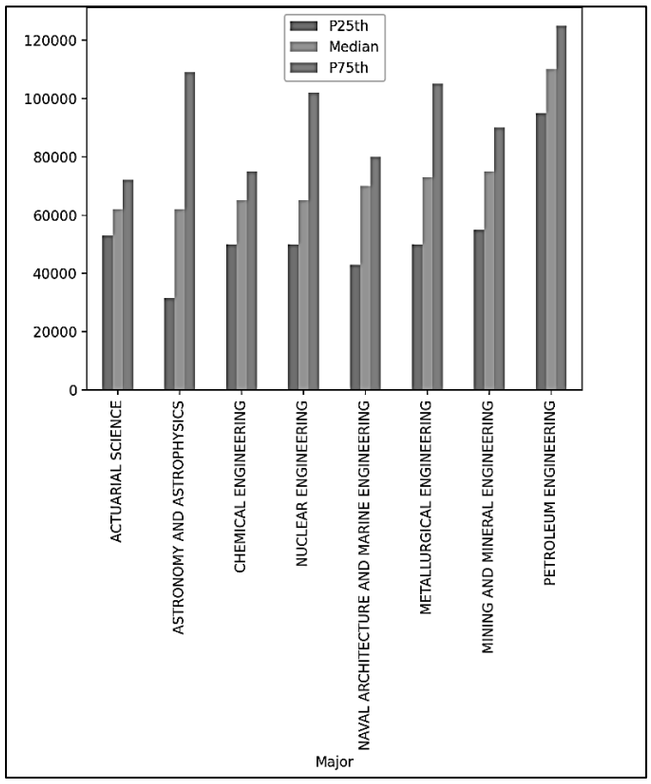
This plot shows that the median salary of petroleum engineering majors is more than $20,000 higher than the rest. The earnings for the second- through fourth-place majors are relatively close to one another.

If you have a data point with a much higher or lower value than the rest, then you'll probably want to investigate a bit further. For example, you can look at the columns that contain related data.

Let's investigate all majors whose median salary is above $60,000. First, you need to filter these majors with the mask df[df["Median"] > 60000]. Then you can create another bar plot showing all three earnings columns:

```
top_medians = df[df["Median"] > 60000].sort_values("Median")
```

```
top_medians.plot(x="Major", y=["P25th", "Median", "P75th"], kind="bar")
```

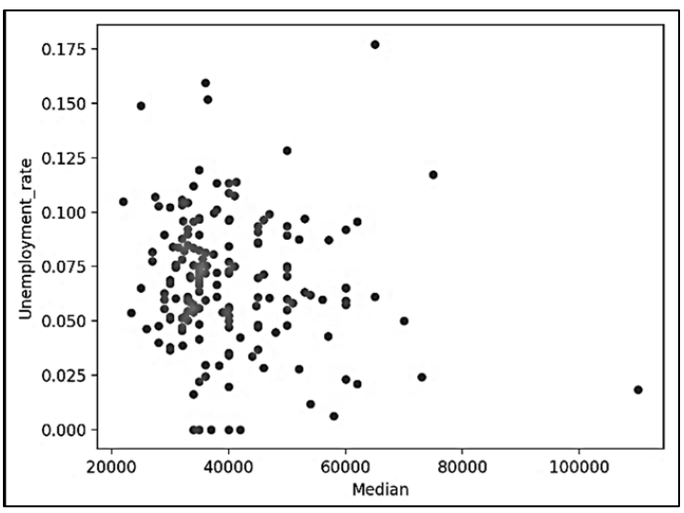


The 25th and 75th percentile confirm what you've seen above: petroleum engineering majors were by far the best paid recent graduates.

Check for Correlation

Often you want to see whether two columns of a dataset are connected. If you pick a major with higher median earnings, do you also have a lower chance of unemployment? As a first step, create a scatter plot with those two columns:

```
df.plot(x="Median", y="Unemployment_rate", kind="scatter")
```



A quick glance at this figure shows that there's no significant correlation between the earnings and unemployment rate.

While a scatter plot is an excellent tool for getting a first impression about possible correlation, it certainly isn't definitive proof of a connection. For an overview of the correlations between different columns, you can use .corr(). If you suspect a correlation between two values, then you have several tools at your disposal to verify your hunch and measure how strong the correlation is.

Keep in mind, though, that even if a correlation exists between two values, it still doesn't mean that a change in one would result in a change in the other. In other words, correlation does not imply causation.

Analyze Categorical Data

Grouping

A basic usage of categories is grouping and aggregation. You can use .groupby() to determine how popular each of the categories in the college major dataset are:

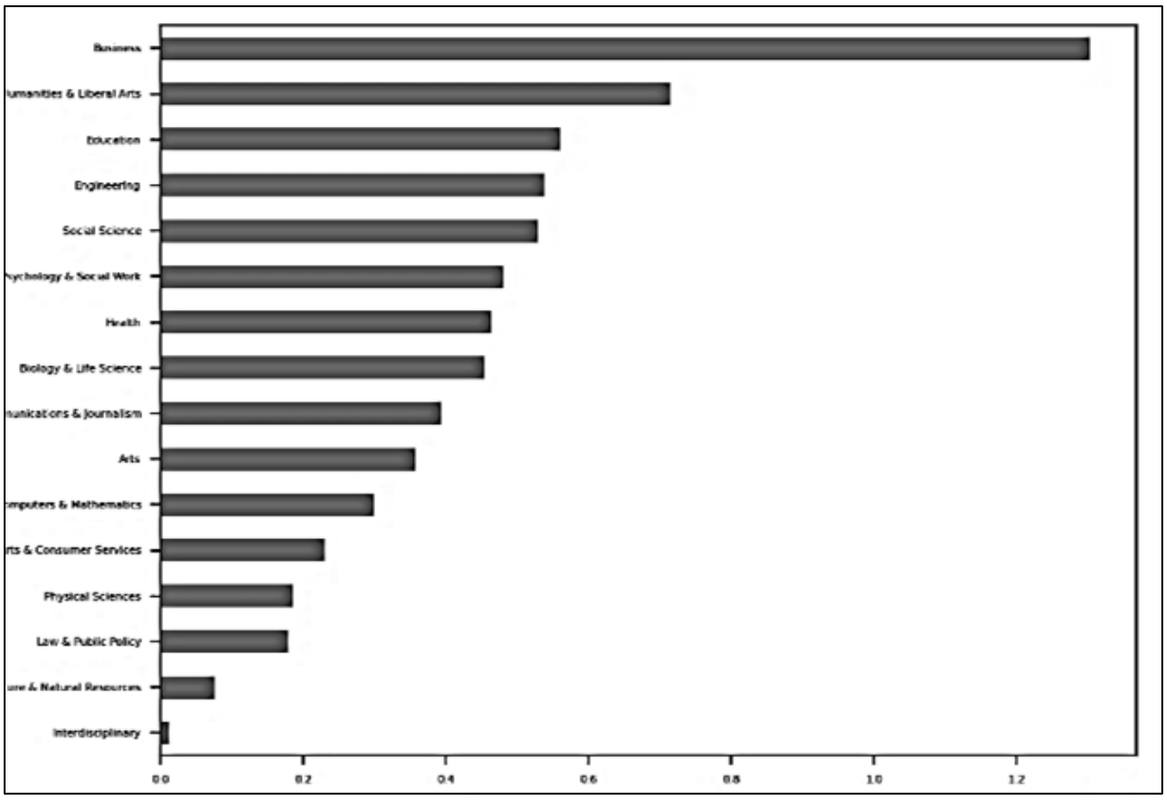
```
In [20]: cat_totals = df.groupby("Major_category")["Total"].sum().sort_values()

In [21]: cat_totals
Out[21]:
Major_category
Interdisciplinary                      12296.0
Agriculture & Natural Resources        75620.0
Law & Public Policy                   179107.0
Physical Sciences                     185479.0
Industrial Arts & Consumer Services   229792.0
Computers & Mathematics               299008.0
Arts                                  357130.0
Communications & Journalism           392601.0
Biology & Life Science                453862.0
Health                                463230.0
Psychology & Social Work              481007.0
Social Science                        529966.0
Engineering                           537583.0
Education                             559129.0
Humanities & Liberal Arts             713468.0
Business                             1302376.0
Name: Total, dtype: float64
```

With .groupby(), you create a DataFrameGroupBy object. With .sum(), you create a Series.

horizontal bar plot showing all the category totals in cat_totals:

```
cat_totals.plot(kind="barh", fontsize=4)
```



Determining Ratios

Vertical and horizontal bar charts are often a good choice if you want to see the difference between your categories. If you're interested in ratios, then pie plots are an excellent tool. However, since cat_totals contains a few smaller categories, creating a pie plot with cat_totals.plot(kind="pie") will produce several tiny slices with overlapping labels .

To address this problem, you can lump the smaller categories into a single group. Merge all categories with a total under 100,000 into a category called "Other", then create a pie plot:

```
small_cat_totals = cat_totals[cat_totals < 100_000]
```

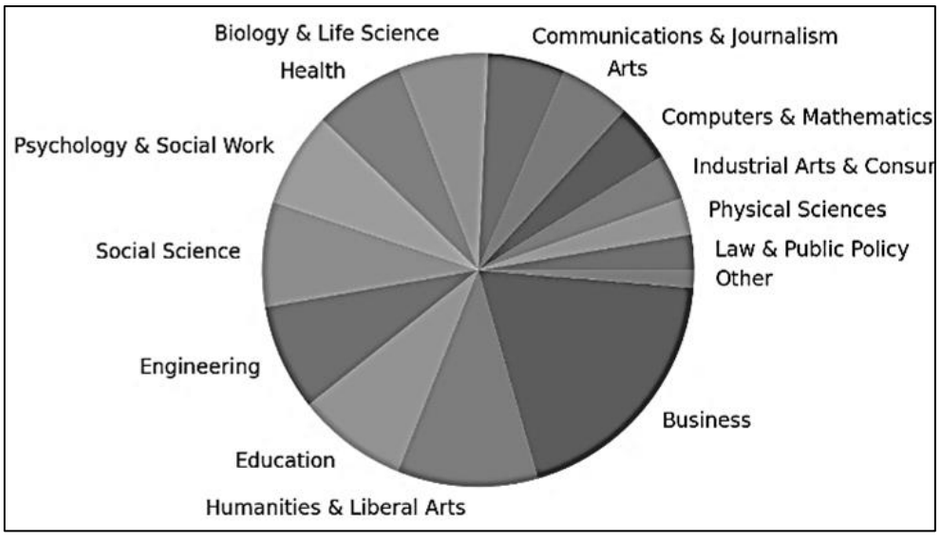
```
big_cat_totals = cat_totals[cat_totals > 100_000]
```

```
# Adding a new item "Other" with the sum of the small categories
small_sums = pd.Series([small_cat_totals.sum()], index=["Other"])
```

```
big_cat_totals = big_cat_totals.append(small_sums)
```

```
big_cat_totals.plot(kind="pie", label="")
```

Notice that you include the argument label="". By default, pandas adds a label with the column name. That often makes sense, but in this case it would only add noise.

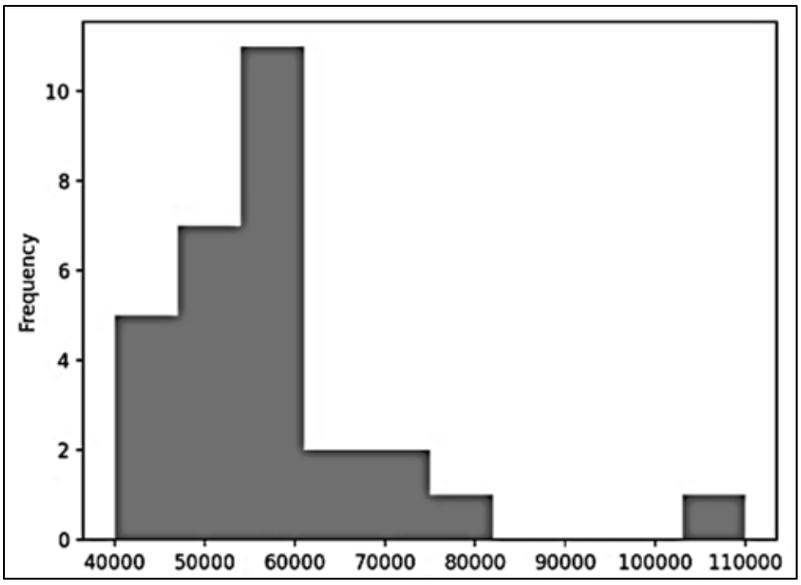


Zooming in on Categories

Sometimes you also want to verify whether a certain categorization makes sense. Are the members of a category more similar to one other than they are to the rest of the dataset? Again, a distribution is a good tool to get a first overview. Generally, we expect the distribution of a category to be similar to the normal distribution but have a smaller range.

Create a histogram plot showing the distribution of the median earnings for the engineering majors:

```
df[df["Major_category"] == "Engineering"]["Median"].plot(kind="hist")
```



You'll get a histogram that you can compare to the histogram of all majors from the beginning

The range of the major median earnings is somewhat smaller, starting at $40,000. The distribution is closer to normal, although its peak is still on the left. So, even if you've decided to pick a major in the engineering category, it would be wise to dive deeper and analyze your options more thoroughly.

View and Select Data Demo

Indexing

importing the necessary libraries

```
import pandas as pd
import numpy as np
```

creating a time series using the date_range method from pandas

```
time_series = pd.date_range('1/1/2020', periods=20)

df = pd.DataFrame(np.random.randn(20, 5),                # 20 refers to the number of rows
                                                         # 5 refers to the number of columns
        index=time_series,
        columns=['Column 1', 'Column 2', 'Column 3', 'Column 4', 'Column 5'])
```

lets check the dataframe that we have just created

```
df
```

	Column 1	Column 2	Column 3	Column 4	Column 5
2020-01-01	-0.506672	-0.223504	1.374367	-0.673617	-1.594992
2020-01-02	-0.288263	-0.308458	0.118085	0.552600	0.057227
2020-01-03	0.208609	0.340054	1.385632	-1.391342	1.447169
2020-01-04	0.755634	-0.584098	1.462399	0.169597	0.646324
2020-01-05	-2.071256	-0.095079	0.967895	-0.553546	-1.772507
2020-01-06	-0.553698	1.113536	-0.208415	-0.971505	0.448935
2020-01-07	2.179848	1.115860	-0.742047	0.039219	1.084622
2020-01-08	1.773190	-0.763939	0.731432	0.760961	-1.826588
2020-01-09	-1.410691	-0.435762	-0.312730	-0.420567	1.313196
2020-01-10	-2.198756	-0.140123	-0.278568	1.416964	-2.059844
2020-01-11	-0.367993	1.378564	-1.204453	-0.473113	-0.794758
2020-01-12	1.143618	0.460143	0.367623	0.319697	0.575761
2020-01-13	0.316275	0.946194	1.169399	-0.609133	0.136591
2020-01-14	-1.233144	0.239735	-1.435708	0.684846	1.075830
2020-01-15	1.369500	1.410637	0.092005	-0.044152	-0.193836
2020-01-16	-0.822112	-0.209220	-1.553531	1.470538	-0.962117
2020-01-17	-0.101630	-0.394767	0.098769	-1.416084	1.107761
2020-01-18	-1.573445	1.087323	-1.315079	0.293028	-0.680757
2020-01-19	-0.658735	0.197481	-0.258119	-1.348261	-0.546297
2020-01-20	0.400129	0.635901	-0.002345	-1.364261	-0.305020

```
df['Column 1']
```

```
2020-01-01   -0.506672
2020-01-02   -0.288263
2020-01-03    0.208609
2020-01-04    0.755634
2020-01-05   -2.071256
2020-01-06   -0.553698
2020-01-07    2.179848
2020-01-08    1.773190
2020-01-09   -1.410691
```

```
df[['Column 1', 'Column 2', 'Column 3']]
```

	Column 1	Column 2	Column 3
2020-01-01	-0.506672	-0.223504	1.374367
2020-01-02	-0.288263	-0.308458	0.118085
2020-01-03	0.208609	0.340054	1.385632
2020-01-04	0.755634	-0.584098	1.462399
2020-01-05	-2.071256	-0.095079	0.967895
2020-01-06	-0.553698	1.113536	-0.208415
2020-01-07	2.179848	1.115860	-0.742047
2020-01-08	1.773190	-0.763939	0.731432
2020-01-09	-1.410691	-0.435762	-0.312730
2020-01-10	-2.198756	-0.140123	-0.278568
2020-01-11	-0.367993	1.378564	-1.204453
2020-01-12	1.143618	0.460143	0.367623

timeseries specific indexing

```
col1 = df['Column 1']
col1[time_series[3]]
[output] 0.755634442895147
```

Slicing

*access sequences

iloc attribute access a group of rows and columns by index location
print first 5 rows of column 1 and column 2

```
df.iloc[:5, 0:2]
```

	Column 1	Column 2
2020-01-01	-0.506672	-0.223504
2020-01-02	-0.288263	-0.308458
2020-01-03	0.208609	0.340054
2020-01-04	0.755634	-0.584098
2020-01-05	-2.071256	-0.095079

Striding
Print all columns, with a 3 step stride by adding additional colon and specifying the step.

```
df[::3]
```

	Column 1	Column 2	Column 3	Column 4	Column 5
2020-01-01	-0.506672	-0.223504	1.374367	-0.673617	-1.594992
2020-01-04	0.755634	-0.584098	1.462399	0.169597	0.646324
2020-01-07	2.179848	1.115860	-0.742047	0.039219	1.084622
2020-01-10	-2.198756	-0.140123	-0.278568	1.416964	-2.059844
2020-01-13	0.316275	0.946194	1.169399	-0.609133	0.136591
2020-01-16	-0.822112	-0.209220	-1.553531	1.470538	-0.962117
2020-01-19	-0.658735	0.197481	-0.258119	-1.348261	-0.546297

Filtering

```
df[(df['Column 3'] < 0)]
```

	Column 1	Column 2	Column 3	Column 4	Column 5
2020-01-06	-0.553698	1.113536	-0.208415	-0.971505	0.448935
2020-01-07	2.179848	1.115860	-0.742047	0.039219	1.084622
2020-01-09	-1.410691	-0.435762	-0.312730	-0.420567	1.313196
2020-01-10	-2.198756	-0.140123	-0.278568	1.416964	-2.059844
2020-01-11	-0.367993	1.378564	-1.204453	-0.473113	-0.794758
2020-01-14	-1.233144	0.239735	-1.435708	0.684846	1.075830
2020-01-16	-0.822112	-0.209220	-1.553531	1.470538	-0.962117
2020-01-18	-1.573445	1.087323	-1.315079	0.293028	-0.680757
2020-01-19	-0.658735	0.197481	-0.258119	-1.348261	-0.546297
2020-01-20	0.400129	0.635901	-0.002345	-1.364261	-0.305020

```
df[(df['Column 1'] < 0) & (df['Column 2'] > 0)][['Column 4', 'Column 5']]
```

	Column 4	Column 5
2020-01-06	-0.971505	0.448935
2020-01-11	-0.473113	-0.794758
2020-01-14	0.684846	1.075830
2020-01-18	0.293028	-0.680757
2020-01-19	-1.348261	-0.546297

Missing Values

Notes:

- A lot of time are spent in Data Cleaning which includes tasks like cleaning missing values
- Missing values occur when there is no data or value stored for the variable in an observation
- This is quite common and can have significant effect to conclusion
- Examples of when there might be missing values:
    - Incomplete survey forms fields skipped by participants
    - Accidental or data entry errors
    - Pending information
    - Deleted outdated data

Delete missing values?

Notes:

- Sometimes deleting records with missing values is the way to go.
- But we may be left with small dataset and a small dataset implies an inaccurate machine learning model.
- We can delete the columns or rows where the percentage of missing values is very high
    - If the column has around 30-40% (or more than that) then we can delete such columns
    - If the dataset has more than 50 columns, we can delete rows if the row has more than 10 values missing, hence 20% or more missing values from rows we can delete rows.
- These are just recommendations and can vary from scenarios
- Check before deleting the columns:
    - If a column has high correlation with the target column
    - If the removal of a row or column can lead to information loss

Filling up missing values

- Business logic
  Ex: missing n-star ratings for products
  -here we do not input 0, will lead to very low rating
  -we can use negative numbers like -1 to -999
- Statistical
  Ex. Mean / Median / Mode

Activity: Interpolate Missing Values in Pandas

Suppose we have the following pandas DataFrame that shows the total sales made by a store during 15 consecutive days:

```
import pandas as pd
import numpy as np
```

#create DataFrame

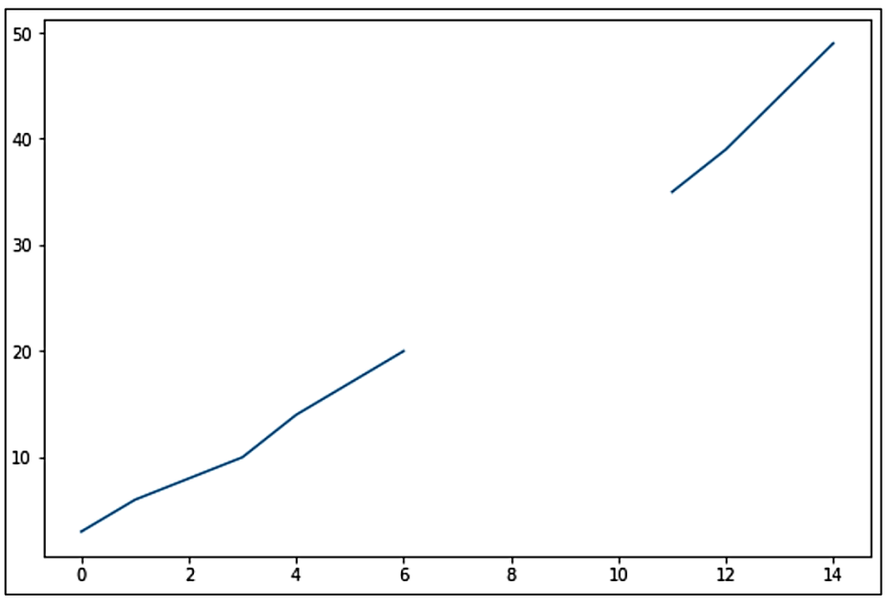
```
df = pd.DataFrame({'day': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15],
                   'sales': [3, 6, 8, 10, 14, 17, 20, np.nan, np.nan, np.nan, np.nan, 35, 39, 44, 49]})
```

Notice that we're missing sales numbers for four days in the data frame.

#create line chart to visualize sales

```
df['sales'].plot()
```

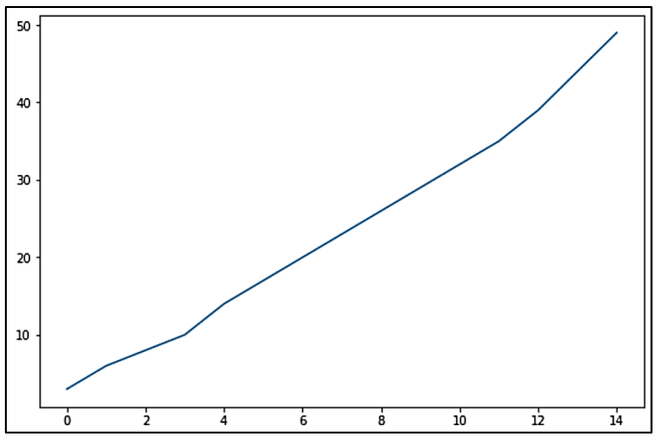
	day	sales
0	1	3.0
1	2	6.0
2	3	8.0
3	4	10.0
4	5	14.0
5	6	17.0
6	7	20.0
7	8	NaN
8	9	NaN
9	10	NaN
10	11	NaN
11	12	35.0
12	13	39.0
13	14	44.0
14	15	49.0



#interpolate missing values in 'sales' column

```
df['sales'] = df['sales'].interpolate()
df
```

	day	sales
0	1	3.0
1	2	6.0
2	3	8.0
3	4	10.0
4	5	14.0
5	6	17.0
6	7	20.0
7	8	23.0
8	9	26.0
9	10	29.0
10	11	32.0
11	12	35.0
12	13	39.0
13	14	44.0
14	15	49.0



Activity: Imputing Data with Pandas

Wine Magazine Dataset: https://www.kaggle.com/datasets/christopheiv/winemagdata130k

import libraries and read dataset

```
import pandas as pd
df = pd.read_csv("winemag-data-130k-v2.csv")
```

display some data, observe columns and missing values

```
df.head()
```

Display missing value information like the number of non-null values in each column:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 129971 entries, 0 to 129970
Data columns (total 14 columns):
 #   Column                 Non-Null Count   Dtype
---  ------                 --------------   -----
 0   Unnamed: 0             129971 non-null  int64
 1   country                129908 non-null  object
 2   description            129971 non-null  object
 3   designation            92506 non-null   object
 4   points                 129971 non-null  int64
 5   price                  120975 non-null  float64
 6   province               129908 non-null  object
 7   region_1               108724 non-null  object
 8   region_2               50511 non-null   object
 9   taster_name            103727 non-null  object
 10  taster_twitter_handle  98758 non-null   object
 11  title                  129971 non-null  object
 12  variety                129970 non-null  object
 13  winery                 129971 non-null  object
dtypes: float64(1), int64(2), object(11)
memory usage: 13.9+ MB
```

calculate the number of missing values in each column:

```
df.isnull().sum()
```

```
Unnamed: 0                  0
country                    63
description                 0
designation             37465
points                      0
price                    8996
province                   63
region_1                21247
region_2                79460
taster_name             26244
taster_twitter_handle   31213
title                       0
variety                     1
winery                      0
dtype: int64
```

The 'price' column contains 8996 missing values.
We can replace these missing values using the '.fillna()' method.
For example, let's fill in the missing values with the mean price:

```
df['price'].fillna(df['price'].mean(), inplace = True)
df.isnull().sum()
```

```
Unnamed: 0                  0
country                    63
description                 0
designation             37465
points                      0
price                       0
province                   63
region_1                21247
region_2                79460
taster_name             26244
taster_twitter_handle   31213
title                       0
variety                     1
winery                      0
dtype: int64
```

We see that the 'price' column no longer has missing values.

Now, suppose we wanted to make a more accurate imputation.
A good guess would be to replace missing values in the price column with the mean prices within the countries the missing values belong.

For example, if we consider missing wine prices for Italian wine, we can replace these missing values with the mean price of Italian wine.

To proceed, let's look at the distribution in 'country' values. We can use the 'Counter' method from the collections module to do so:

```
from collections import Counter
print(Counter(df['country']))
```

```
Counter({'US': 54504, 'France': 22093, 'Italy': 19540, 'Spain': 6645, 'Portugal': 5691, 'Chile': 4472, 'Argentina': 3800, 'Aust
ria': 3345, 'Australia': 2329, 'Germany': 2165, 'New Zealand': 1419, 'South Africa': 1401, 'Israel': 505, 'Greece': 466, 'Canad
a': 257, 'Hungary': 146, 'Bulgaria': 141, 'Romania': 120, 'Uruguay': 109, 'Turkey': 90, 'Slovenia': 87, 'Georgia': 86, 'Englan
d': 74, 'Croatia': 73, 'Mexico': 70, nan: 63, 'Moldova': 59, 'Brazil': 52, 'Lebanon': 35, 'Morocco': 28, 'Peru': 16, 'Ukraine':
14, 'Czech Republic': 12, 'Serbia': 12, 'Macedonia': 12, 'Cyprus': 11, 'India': 9, 'Switzerland': 7, 'Luxembourg': 6, 'Armeni
a': 2, 'Bosnia and Herzegovina': 2, 'Slovakia': 1, 'China': 1, 'Egypt': 1})
```

Let's take a look at wine made in the 'US'. We can define a data frame containing only 'US' wines:

```
df_US = df[df['country']=='US']
```

Now, let's print the number of missing values:

```
df_US.isnull().sum()
```

```
Unnamed: 0                  0
country                     0
description                 0
designation             17596
points                      0
price                     239
province                    0
region_1                  278
region_2                 3993
taster_name             16774
taster_twitter_handle   19763
title                       0
variety                     0
winery                      0
dtype: int64
```

We see that there are 239 missing 'price' values in the 'US' wines data. To fill in the missing values with the mean corresponding to the prices in the US we do the following:

```
df_US['price'].fillna(df_US['price'].mean(), inplace = True)
```

Now suppose we wanted to do this for the missing price values in each country. First, let's impute missing country values:

```
df['country'].fillna(df['country'].mode()[0], inplace = True)
```

Next, within a for-loop we can define country-specific data frames:

```
for i in list(set(df['country'])):
    df_country = df[df['country']== country]
```

Next, we can fill the missing values in these country-specific data frames with their respective mean prices:

```
for i in list(set(df['country'])):
    df_country = df[df['country']== country]
    df_country['price'].fillna(df_country['price'].mean(),inplace = True)
```

We then append the result to a list we'll call "frames"

```
frames = []
for i in list(set(df['country'])):
    df_country = df[df['country']== country]
    df_country['price'].fillna(df_country['price'].mean(),inplace = True)
    frames.append(df_country)
```

Finally, we concatenate the resulting list of data frames:

```
frames = []
for i in list(set(df['country'])):
    df_country = df[df['country']== i]
    df_country['price'].fillna(df_country['price'].mean(),inplace = True)
    frames.append(df_country)
    final_df = pd.concat(frames)
```

Now, if we print the number of missing price values before imputation we get:

```
print(df.isnull().sum())
```

```
country                    63
description                 0
designation             37465
points                      0
price                    8996
province                   63
region_1                21247
region_2                79460
taster_name             26244
taster_twitter_handle   31213
title                       0
variety                     1
winery                      0
```

And after imputation:

```
print(final_df.isnull().sum())
```

```
country                     0
description                 0
designation             37454
points                      0
price                       1
province                    0
region_1                21184
region_2                79397
taster_name             26244
taster_twitter_handle   31213
title                       0
variety                     1
winery                      0
```

We see all but one of the missing values have been imputed. This corresponds to wines in Egypt which has no price data.

We can fix this by checking the length of the data frame within the for loop and only imputing with the country-specific mean if the length is greater than one. If the length is equal to 1 we impute with the mean across all countries:

```
frames = []
for i in list(set(df['country'])):
   df_country = df[df['country']== i]
   if len(df_country) > 1:
      df_country['price'].fillna(df_country['price'].mean(),inplace = True)
   else:
      df_country['price'].fillna(df['price'].mean(),inplace = True)
   frames.append(df_country)
   final_df = pd.concat(frames)
```

Printing the result we see that all of the values have been imputed for the ‘price’ column:

```
final_df.isnull().sum()
```

We can define a function that generalizes this logic. Our function will take variables corresponding to a numerical column and categorical column:

```
def impute_numerical(categorical_column, numerical_column):
   frames = []
   for i in list(set(df[categorical_column])):
      df_category = df[df[categorical_column]== i]
      if len(df_category) > 1:
         df_category[numerical_column].fillna(df_category[numerical_column].mean(),inplace = True)
      else:
         df_category[numerical_column].fillna(df[numerical_column].mean(),inplace = True)
      frames.append(df_category)
      final_df = pd.concat(frames)
   return final_df
```

We perform imputation using our function by executing the following:

```
impute_price  = impute_numerical('country', 'price')
impute_price.isnull().sum()
```

Let’s also verify that the shapes of the original and imputed data frames match

```
print("Original Shape: ", df.shape)
print("Imputed Shape: ", impute_price.shape)
```

```
Original Shape:  (129971, 13)
Imputed Shape:  (129971, 13)
```

Similarly, we can define a function that imputes categorical values. This function will take two variables corresponding columns with categorical values.

```
def impute_categorical(categorical_column1, categorical_column2):
   cat_frames = []
   for i in list(set(df[categorical_column1])):
      df_category = df[df[categorical_column1]== i]
      if len(df_category) > 1:
         df_category[categorical_column2].fillna(df_category[categorical_column2].mode()[0],inplace = True)
      else:
         df_category[categorical_column2].fillna(df[categorical_column2].mode()[0],inplace = True)
      cat_frames.append(df_category)
      cat_df = pd.concat(cat_frames)
   return cat_df
```

We can impute missing 'taster_name' values with the mode in each respective country:

```
impute_taster = impute_categorical('country', 'taster_name')
impute_taster.isnull().sum()
```

We see that the 'taster_name' column now has zero missing values. Again, let's verify that the shape matches with the original data frame:

```
print("Original Shape: ", df.shape)
print("Imputed Shape: ", impute_taster.shape)
```

Join

Join operations are the most used operations using SQL because it mainly helps in making new data using two or more data using one variable. We join data in different the following manners:

Inner join

This join provides common values from the variable on which we decided to join, using the SQL we can perform this operation in the following manner:

```
SELECT *
FROM df1
INNER JOIN df2
  ON df1.key = df2.key;
```

Where we have two data frames(df1 and df2) and one common variable(key). To perform this operation we are required to have two or more datasets. We can make a data frame in Pandas using the following way:

```
import numpy as np
df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],
                    'value': np.random.randn(4)})


df2 = pd.DataFrame({'key': ['B', 'D', 'D', 'E'],
                    'value': np.random.randn(4)})
```

Now we can inner join datasets in the following four ways as given below.

```
pd.merge(df1, df2, on='key')
```

Left outer join

This operation helps us join the datasets using a clause. With the help of this, we can preserve the unmatched rows of the left data and join them with a NULL row in the shape of the right table. In SQL we can perform this operation in the following way:

```
SELECT *
FROM df1
LEFT OUTER JOIN df2
  ON df1.key = df2.key;
```

the same operation can be performed using pandas in the following way:

```
pd.merge(df1, df2, on='key', how='left')
```

Right outer join

This operation helps us in joining data using a clause using which we preserve rows from the right data and join them with a null in the shape of the first (left) table. Using the SQL we can perform this operation in the following way:

```
SELECT *
FROM df1
RIGHT OUTER JOIN df2
  ON df1.key = df2.key;
```

The same operation Using the Pandas can be performed in the following way

```
pd.merge(df1, df2, on='key', how='right')
```

Full join

This operation preserves all the rows of every data while joining them. This operation can be performed using the SQL in the following way:

```
SELECT *
FROM df1
FULL OUTER JOIN df2
  ON df1.key = df2.key;
```

#This same operation can be performed using Pandas in the following way

```
pd.merge(df1, df2, on='key', how='outer')
```

More Demonstrations

```
import pandas as pd
```

```
# read and display from csv
data = pd.read_csv("CoffeeChain.csv")
data
```

```
# read from excel
data_xls = pd.read_excel("CoffeeChain.xls")
data_xls
```

```
data.head(2)
data.tail(3)
data.count()
data[['Product','Type','Sales']]
data.sample(3)
data[['Product','Type']].value_counts()
```

```
data['Market'].drop_duplicates()
```

```
import numpy as np
markets = list(np.unique(data['Market']))
markets
```

```
# plot to graph
data['Market'].value_counts().plot(kind='bar'
                ,title='Coffee Chain Market Distribution'
                ,xlabel='Market',ylabel='Order Count')
```

```
data['Product'].value_counts().plot(kind='pie',title='Product Distribution')
```

```
# read online dataset
url = "https://raw.githubusercontent.com/johnreygoh/datasets/master/employee.csv"
ol_data = pd.read_csv(url)
ol_data
```

```
data['Market'].drop_duplicates()
```

```
# number of unique entries per column
data.nunique()
```

```
mgroup = data.groupby('Product').sum()
# mgroup['Sales']

# figsize is figure size in inches (width,height)
anno = mgroup.plot(y='Sales'
                   ,kind='bar'
                   ,figsize=(12,6)
                   ,fontsize='8'
                   ,title='Product Sales'
                   ,color='red'
                   ,ylabel='Sales'
)

anno.bar_label(anno.containers[0])
```

Module 6: Data Visualization in Python using matplotlib

Matplotlib is one of the most popular Python packages used for data visualization. It is a cross-platform library for making 2D plots from data in arrays.

```
\>pip install matplotlib
```

let's create a simple plot. We will be plotting two lists containing the X, Y coordinates for the plot.

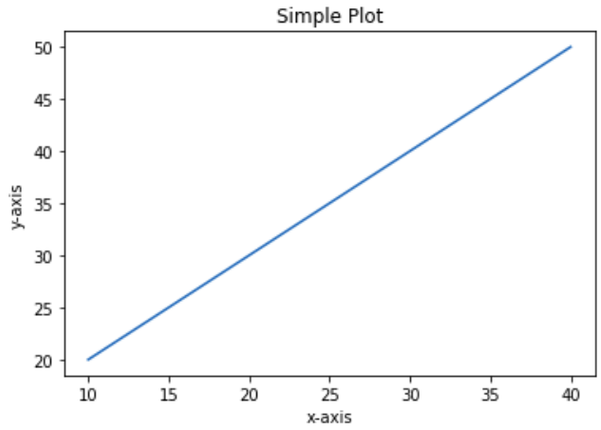
```
import matplotlib.pyplot as plt

# initializing the data
x = [10, 20, 30, 40]
y = [20, 30, 40, 50]

# plotting the data
plt.plot(x, y)

# Adding the title
plt.title("Simple Plot")

# Adding the labels
plt.ylabel("y-axis")
plt.xlabel("x-axis")
plt.show()
```



Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

install seaborn

```
\> pip install seaborn
```

Univariate vs Bivariate vs Multivariate Analysis

- Univariate: Visualize and analyze distribution of a single variable only
- Bivariate: Visualize and analyze distribution of 2 variables, tries to check the relationship between these variables to see if it has associations and the strength of this associations.
- Multivariate: Visualize and analyze distribution of more than 2 variables

Univariate Analysis

#Importing the important libraries

```
import pandas as pd
import seaborn as sns
```

#Reading the dataset

```
data = pd.read_csv('employee.csv')
data.shape
```

out: (1470, 35)

lets check the head of the dataset

```
data.head()
```

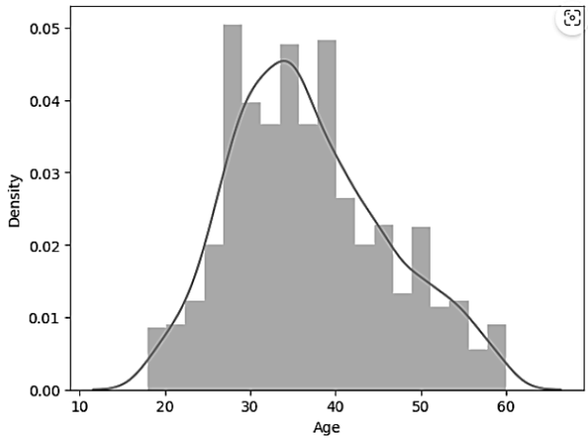
	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	Educati
0	41	Yes	Travel_Rarely	1102	Sales	1	2	Life S
1	49	No	Travel_Frequently	279	Research & Development	8	1	Life S
2	37	Yes	Travel_Rarely	1373	Research & Development	2	2	
3	33	No	Travel_Frequently	1392	Research & Development	3	4	Life S
4	27	No	Travel_Rarely	591	Research & Development	2	1	

5 rows × 35 columns

Univariate Analysis on Numerical Variables

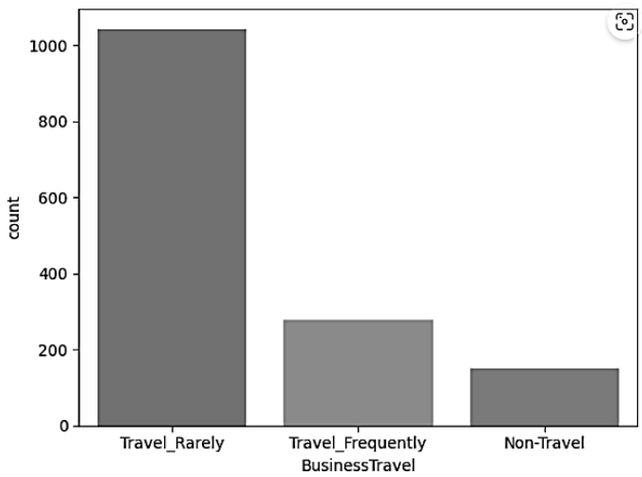
Distribution of Age variable

```
sns.distplot(data['Age'])
```



Univariate Analysis on Categorical Variables

```
sns.countplot(data['BusinessTravel'])
```



Bivariate Analysis

#Importing the important libraries

```
import pandas as pd
import seaborn as sns
```

#Reading the dataset

```
data = pd.read_csv('employee.csv')
data.shape
```

out: (1470, 35)

lets check the head of the dataset

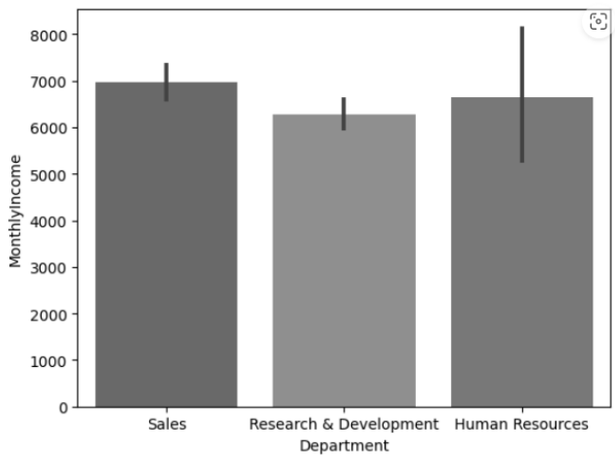
```
data.head()
```

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	Educati
0	41	Yes	Travel_Rarely	1102	Sales	1	2	Life S
1	49	No	Travel_Frequently	279	Research & Development	8	1	Life S
2	37	Yes	Travel_Rarely	1373	Research & Development	2	2	
3	33	No	Travel_Frequently	1392	Research & Development	3	4	Life S
4	27	No	Travel_Rarely	591	Research & Development	2	1	

5 rows × 35 columns

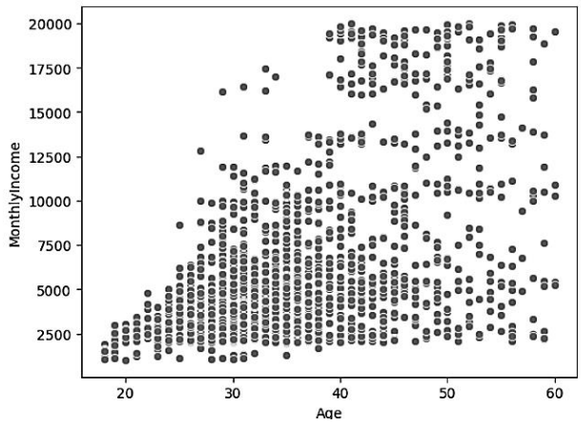
Categorical vs Continuous Variables

```
sns.barplot(x=data['Department'], y=data['MonthlyIncome'])
```



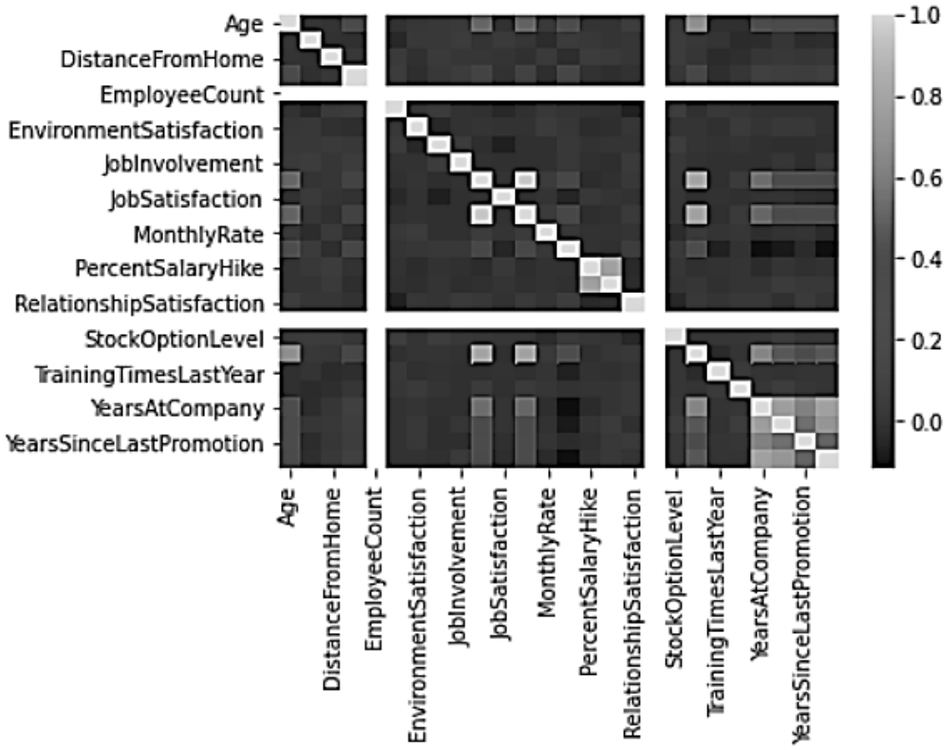
Continous vs Continuous Variables

```
sns.scatterplot(x=data['Age'], y=data['MonthlyIncome'])
```



Multivariate Analysis

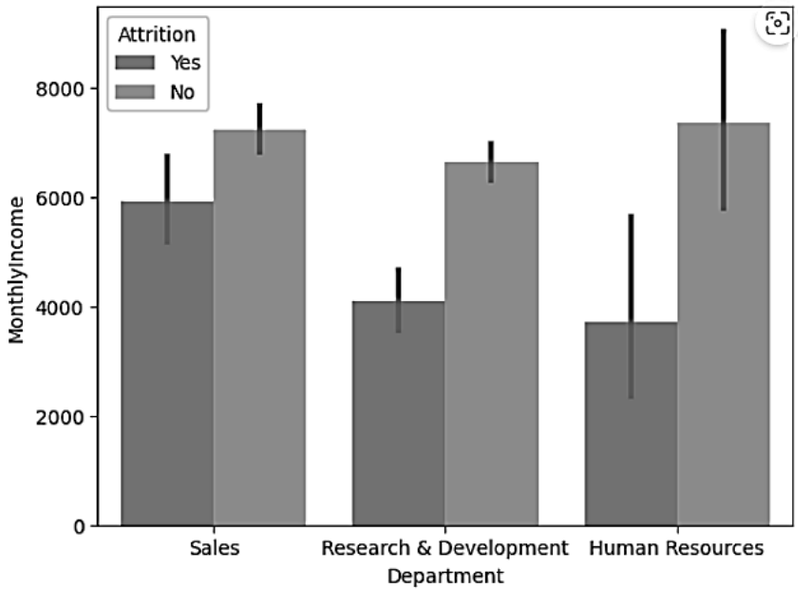
```
sns.heatmap(data.corr())
```



Bar plot with Extra Variable

hue adds a legend

```
sns.barplot(x=data['Department'], y=data['MonthlyIncome'], hue=data['Attrition'])
```



Scatter Plots

Install some modules

```
\> pip install plotly statsmodels
```

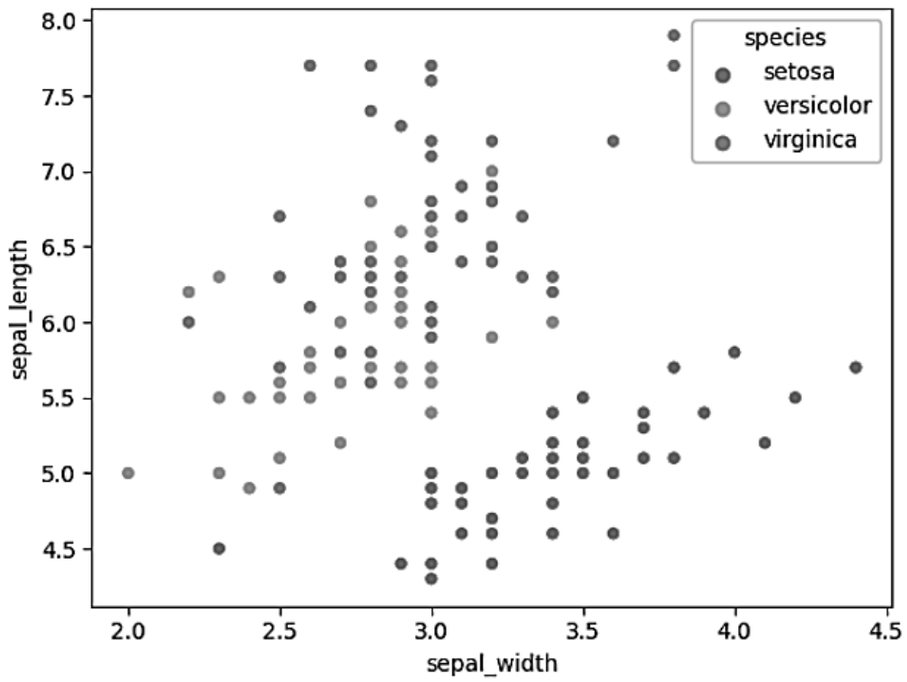
import libs

```
import pandas as pd
import seaborn as sns
import plotly.express as px
```

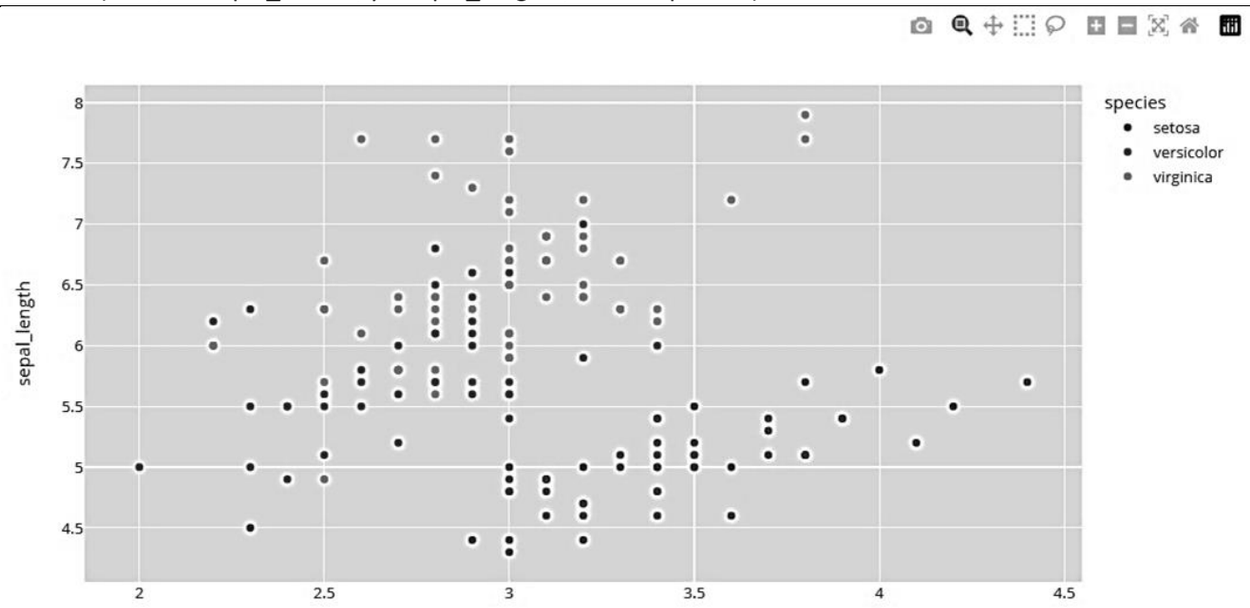
reading the dataset

```
data = px.data.iris()       # iris is a built-in dataset from the plotly data package
data.head()
```

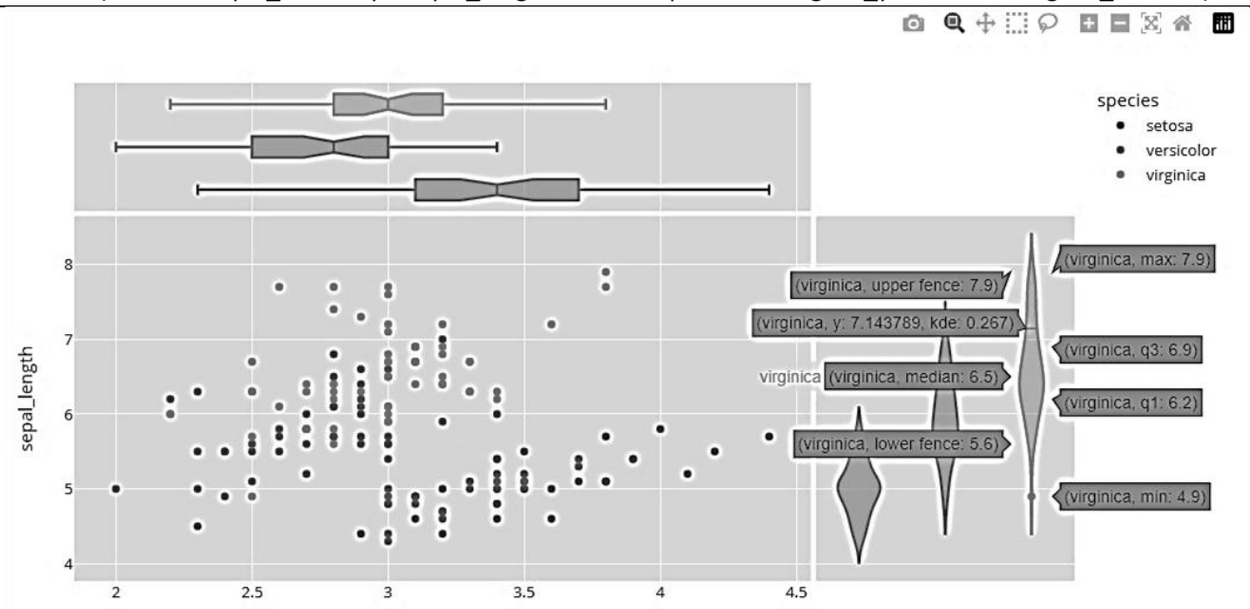
```
sns.scatterplot(data['sepal_width'], data['sepal_length'], hue = data['species'])
```



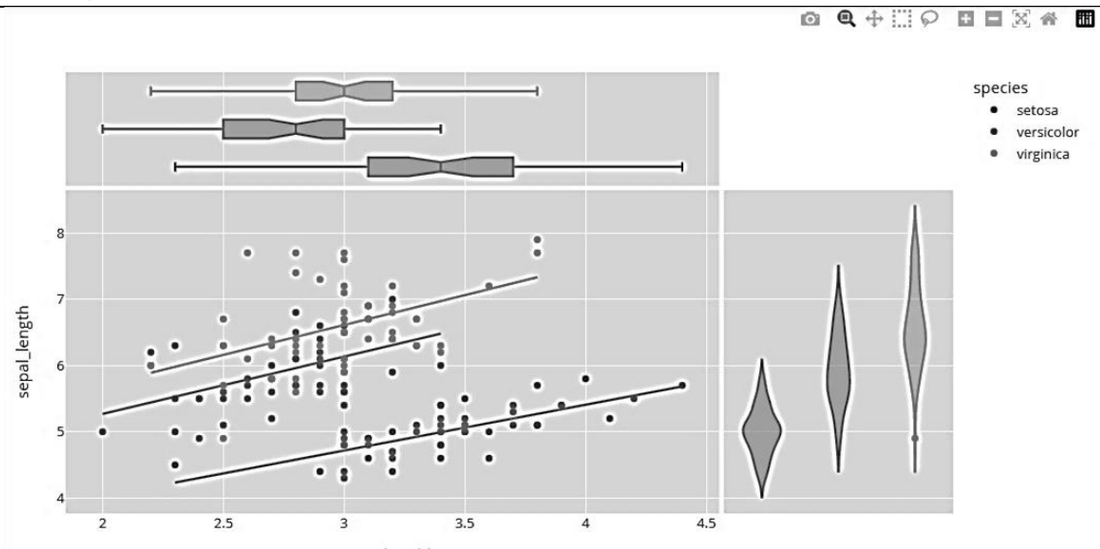
```
px.scatter(data, x="sepal_width", y="sepal_length", color="species")
```




```
px.scatter(data, x="sepal_width", y="sepal_length", color="species", marginal_y="violin", marginal_x="box")
```




```
px.scatter(data, x="sepal_width", y="sepal_length", color="species", marginal_y="violin", marginal_x="box", trendline="ols")
```



Charts with colorscale

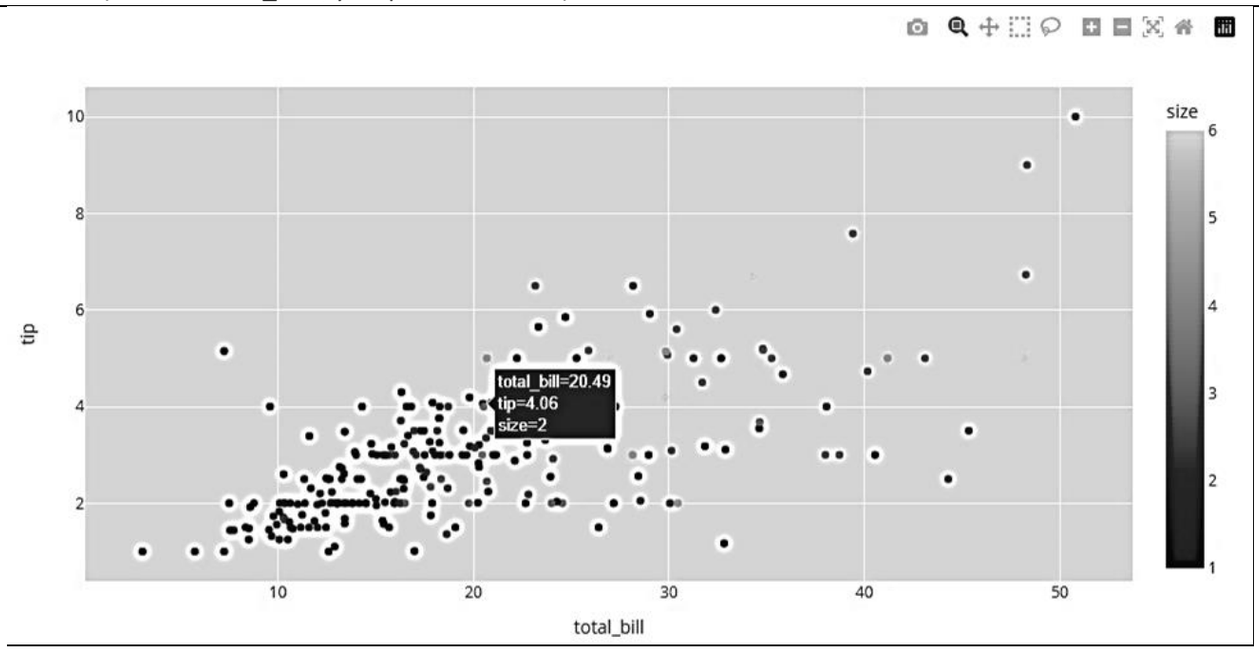
```
import pandas as pd
import plotly.express as px
```

reading the dataset

```
data = px.data.tips()       # tips is a built-in dataset from the plotly data package
data.head()
```

```
px.scatter(data, x="total_bill", y="tip")
```

```
px.scatter(data, x="total_bill", y="tip", color="size")
```



Bar and Line Charts

```
import pandas as pd
import seaborn as sns
import plotly.express as px

data = px.data.gapminder()
data.head()
```

```
sns.lineplot(x=data['year'], y=data['lifeExp'])
```

```
px.line(data, x="year", y="lifeExp")
```

```
px.line(data, x="year", y="lifeExp", color="continent")
```

```
px.line(data, x="year", y="lifeExp", color="continent", line_shape="hv")
```

```
px.line(data, x="year", y="lifeExp", color="continent", line_shape="vh")
```

```
px.area(data, x="year", y="pop", color="continent", line_group="country")
```

```
px.bar(data, x='continent', y='pop')
```

lets read the tips data also

```
df = px.data.tips()
df.head()
```

```
px.bar(df, x="sex", y="total_bill", color="smoker", barmode="group")
```

```
px.bar(df, x="sex", y="total_bill", color="smoker", barmode="stack")
```

Facet Grids

```
import pandas as pd
import plotly.express as px
```

reading the dataset

```
data = px.data.tips()
data.head()
```

```
px.bar(data, x="sex", y="total_bill", color="smoker", barmode="group")
```

```
px.bar(data, x="sex", y="total_bill", color="smoker",
    barmode="group",
    facet_row="time",
    facet_col="day",
    category_orders={"day": ["Thur", "Fri", "Sat", "Sun"], "time": ["Lunch", "Dinner"]})
```

```
px.scatter(data, x="tip", y="total_bill", color="smoker",
    facet_row="time",
    facet_col="day",
    category_orders={"day": ["Thur", "Fri", "Sat", "Sun"], "time": ["Lunch", "Dinner"]})
```

3D Charts

3D Scatter Plot

```
import pandas as pd
import plotly.express as px

# Load the dataset
df = pd.read_csv(url)

# Create a 3D scatter plot
scatter_plot = px.scatter_3d(df, x='Sales', y='Profit', z='Quantity', color='Quantity',
                    title='3D Scatter Plot of Sales, Profit, and Quantity')
# Customize graph settings
scatter_plot.update_layout(title_font_size=20, title_x=0.5,
                    legend_title_font_color="blue",
                    coloraxis_colorbar=dict(title="Quantity"))

scatter_plot.show()
```

Bar plot

For the 3D bar plot, Plotly's standard bar plot (which is 2D) can be used, as Plotly does not have a direct 3D bar plot function. However, you can represent 3D data effectively by aggregating or segmenting your data.

```
import plotly.graph_objects as go

# Aggregating data by Category
category_summary = df.groupby('Category').sum().reset_index()
```

```
# Create a bar plot
bar_plot = go.Figure(data=[go.Bar(x=category_summary['Category'], y=category_summary['Sales'])])
bar_plot.update_layout(title='Total Sales by Category',
                       xaxis_title='Category',
                       yaxis_title='Total Sales',
                       title_font_size=20, title_x=0.5)
# Customize the bar plot
bar_plot.update_traces(marker_color='green', marker_line_color='rgb(8,48,107)',
                       marker_line_width=1.5, opacity=0.6)

bar_plot.show()
```

Maps

```
import plotly.express as px
```

```
df = px.data.carshare()
df.head()
```

```
# Map based on Latitude and Longitude
px.scatter_mapbox(df, lat="centroid_lat",
                  lon="centroid_lon",
                  color="peak_hour",
                  size="car_hours",
                  size_max=15, zoom=10,
                  mapbox_style="carto-positron")
```

```
# Scatter Map
df = px.data.gapminder()
df.head()
```

```
px.scatter_geo(df, locations="iso_alpha",
               color="continent",
               hover_name="country",
               size="pop",
               projection="natural earth")
```

Animations

 # Animated bubbleplots

```
import plotly.express as px
import warnings

# Load the dataset
data = px.data.gapminder()

# Suppress warnings if any
warnings.filterwarnings('ignore')

# Create the bubble plot using Plotly Express
figure = px.scatter(data,
                    x='lifeExp',
                    y='gdpPercap',
                    animation_frame='year',
                    size='pop',
                    color='continent',
                    hover_name='country',
                    log_y=True,  # Log scale for y-axis
                    size_max=60,  # Maximum bubble size
```

```
            title='Life Expectancy vs GDP Per Capita Over Time')

figure.update_layout(height=650)

# Show the plot
figure.show()
```

 # Animation with facets

```
import plotly.express as px

df = px.data.gapminder()
df.head()

px.scatter(df, x="gdpPercap", y="lifeExp",
        animation_frame="year",
        animation_group="country",
        size="pop",
        color="continent",
        hover_name="country",
        log_x=True, size_max=45, range_x=[100,100000], range_y=[25,90])

px.scatter(df, x="gdpPercap", y="lifeExp",
        animation_frame="year",
        animation_group="country",
        size="pop",
        color="continent",
        hover_name="country",
        facet_col="continent",
        log_x=True, size_max=45, range_x=[100,100000], range_y=[25,90])
```

 # Animation with Scattermaps

```
import plotly.express as px

df = px.data.gapminder()
df.head()

figure=px.scatter_geo(df, locations="iso_alpha",
            color="continent",
            hover_name="country",
            size="pop",
            animation_frame="year",
            projection="natural earth")
figure.update_layout(height=650)
```

 # Animation with Chloropleth maps

```
import plotly.express as px
df = px.data.gapminder()
df.head()

figure=px.choropleth(df, locations="iso_alpha",
            color="lifeExp",
            hover_name="country",
            animation_frame="year",
            range_color=[20,80])

figure.update_layout(height=650)
```