

From <http://www.onlineprogramminglessons.com>

These Python mini lessons will teach you all the Python Programming statements you need to know, so you can write 90% of any Python Program.

Lesson 1 Input and Output

Lesson 2 Functions

Lesson 3 Classes

Lesson 4 Operators

Lesson 5 Lists, Sets, Tuples and Dictionaries

Lesson 6 Programming Statements

Lesson 7 File I/O

**Lesson 8 List Comprehension, Iterators, Generators
and Higher Order Functions**

Lesson 9 Recursion

Lesson 10 Regular Expressions

Lesson 11 SQL and SQLite

Lesson 12 Python Project to do

Let's get started!

You first need a Python interpreter to run Python Programs.

Download from this site: <https://www.python.org/downloads/>

Choose Python version 3.6.4 or higher.

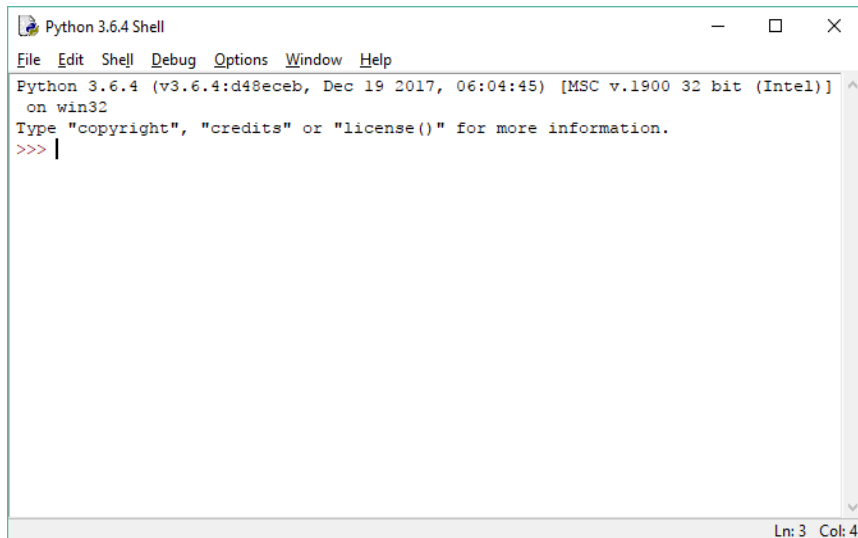
Conventions used in these lessons:

bold - headings, keywords, code

italics - code syntax

underline - important words

Once you download Python and run it you will get this screen known as the Python interpreter shell:

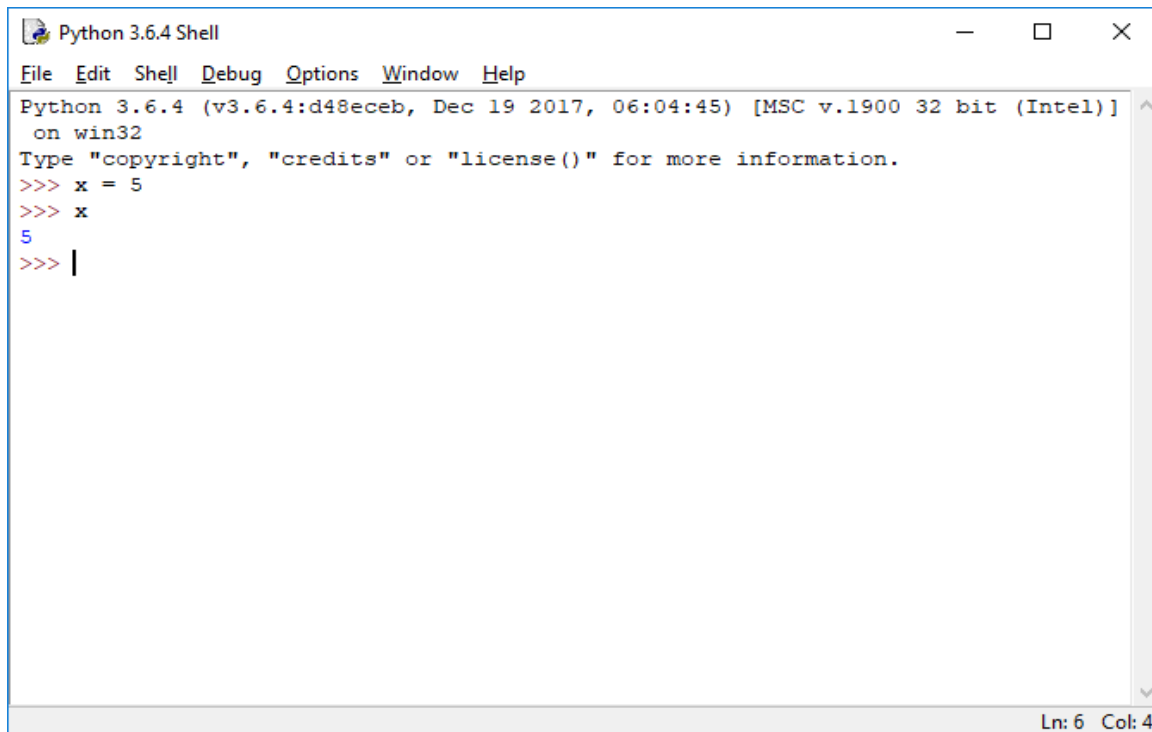


```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> |
Ln: 3 Col: 4
```

Lesson 1 Input and Output

Introduction to variables

Programming is all about storing values and doing operations on them. Values can be numeric numbers like 5 or a decimal number like 10.5 or a text message like "Hello there". Text messages are known as strings and are enclosed in double or single quote. Operation on values maybe adding two values together or joining two strings together. When a Python program is run, values are stored in a computer memory location. In a Python program the memory location is represented by a identifier name. This identifier name is known as a variable. A variable may store many different values at different times as the Python program runs. We now make our first variable and assign a value to it. For convenience *you can do this in the Python interpreter shell that first appears when Python is launched. In the Python shell type `x = 5` and then press the enter key, then type the variable name `x`. You should get something like this:



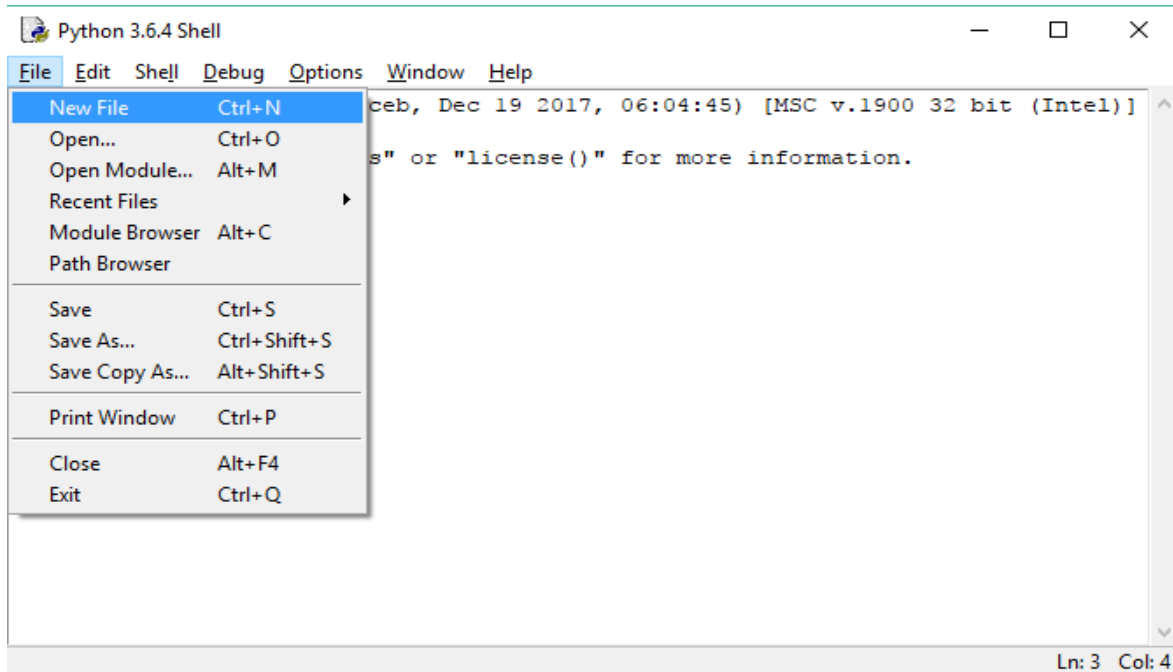
```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> x = 5
>>> x
5
>>> |
```

Ln: 6 Col: 4

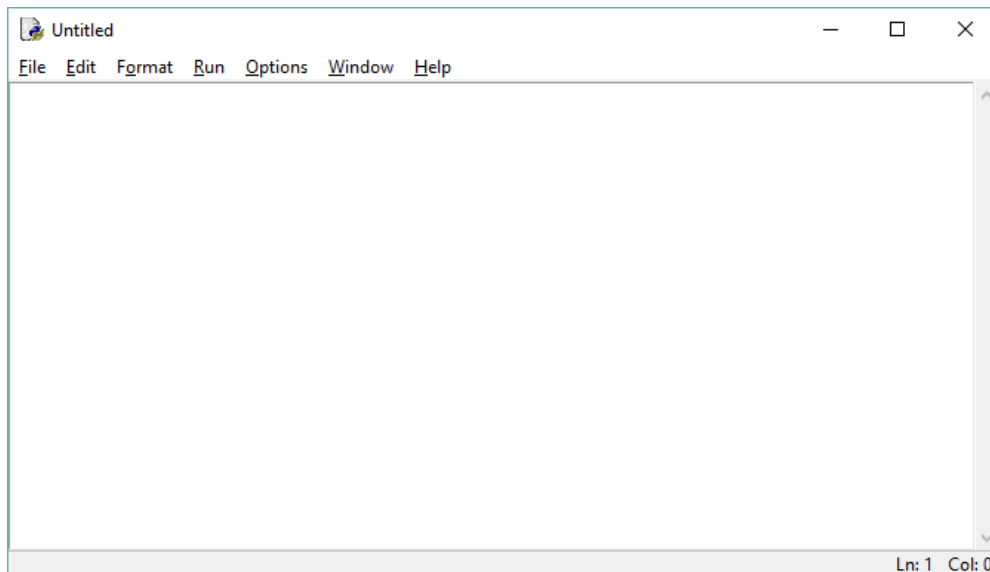
The value 5 is assigned to the variable **x** and **x** now holds the value 5. When you type the variable name **x** in the python interpreter shell the value of the variable is printed out to the screen. When you type **x = 5** in the python interpreter shell **x** stores the value 5. the **x = 5** is known as a programming statement. A program statement is an instruction directing the computer to do operations like store a value, print out a value, get a value from the keyboard or add another value to a variable. A program is just a collection of programming statements.

Programming statements

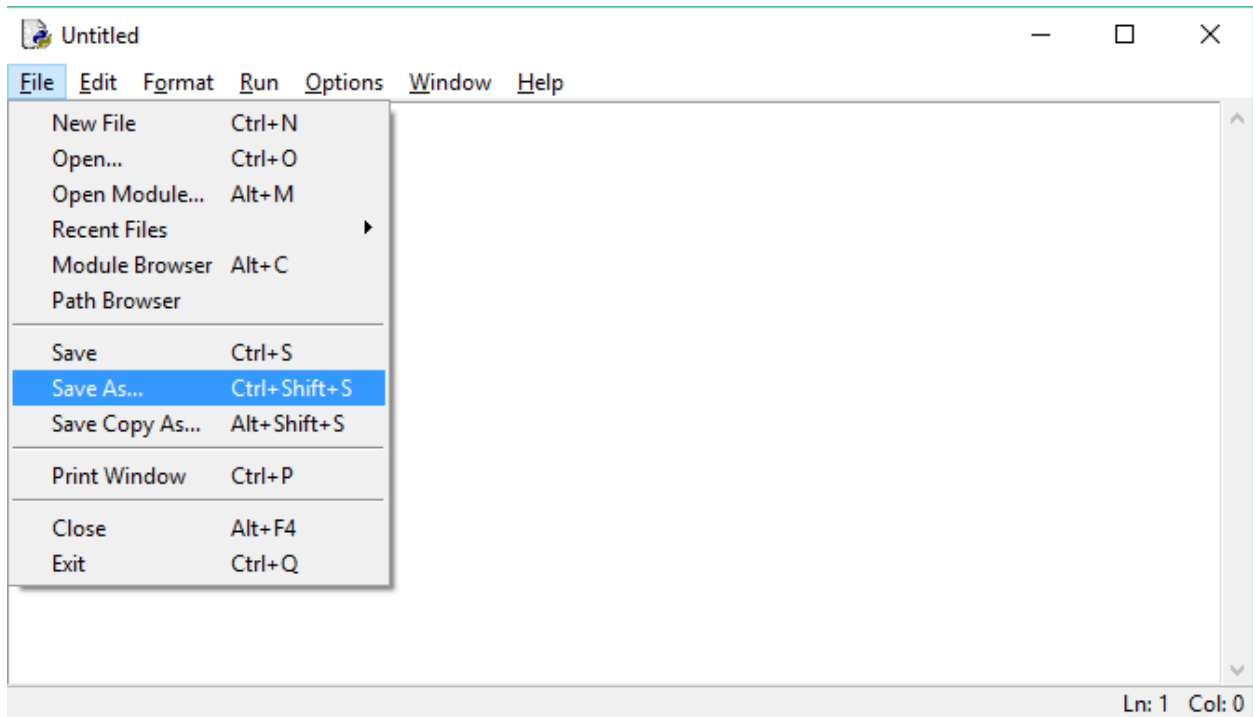
Although you can type python programming statements directly into the Python interpreter shell, but for convenience It is better to store all your lesson programs in a file. A Python program is also known as script, because it is an interpretive language, meaning the Python programming statements are executed one by one as they appear in the program file. To make a Python program file select **New File** from the **File** menu.



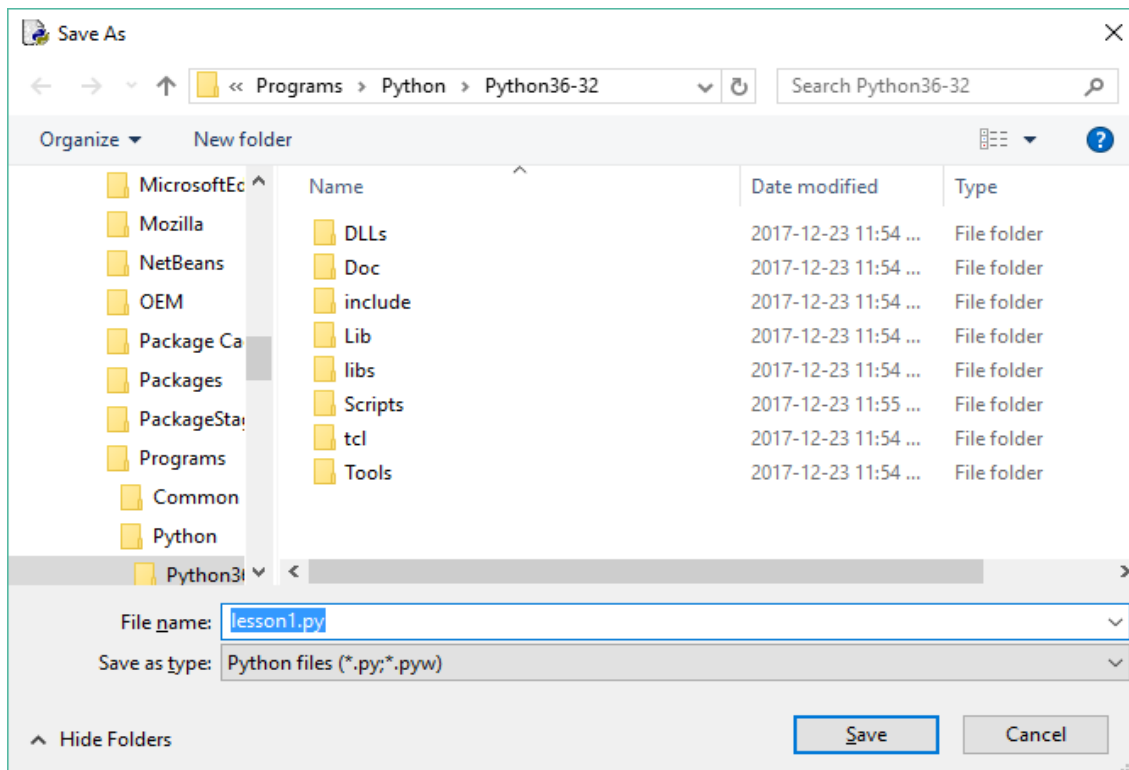
The editor window appears where you can type in Python programming statements that you can save and run. You may want first to make a folder on your computer called python Lessons to store all your python programs.



From the File menu select File Save As

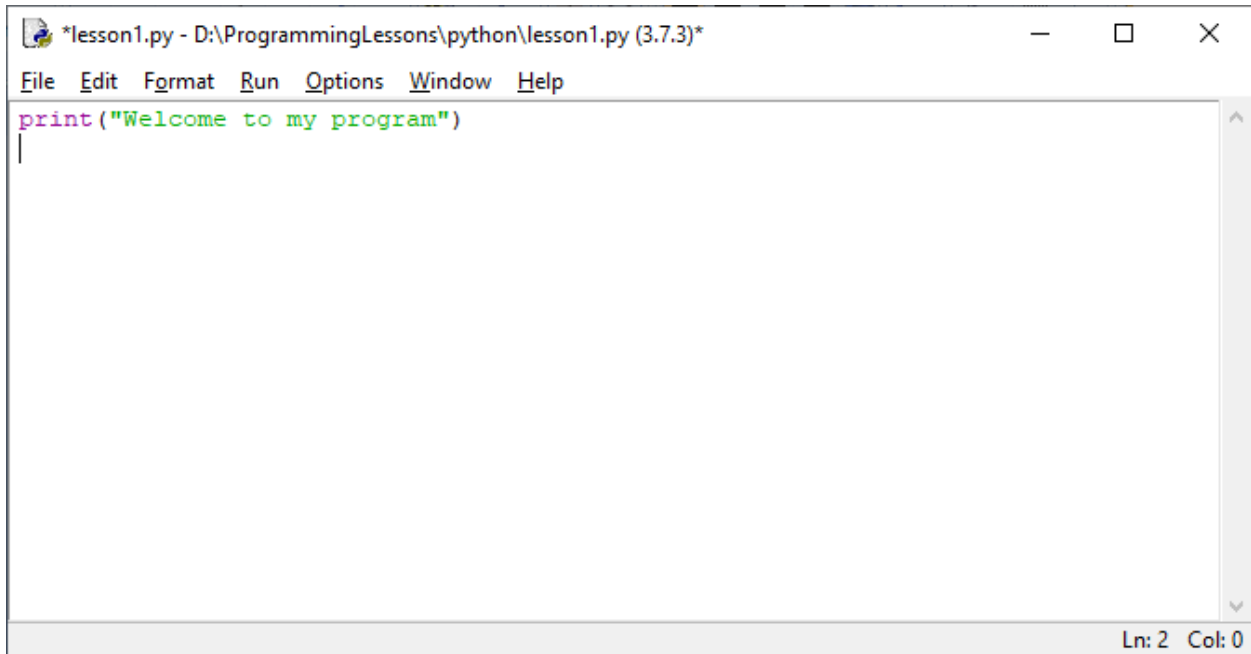


Navigate to your python Lessons folder and save your (empty) python program file as lesson1.py

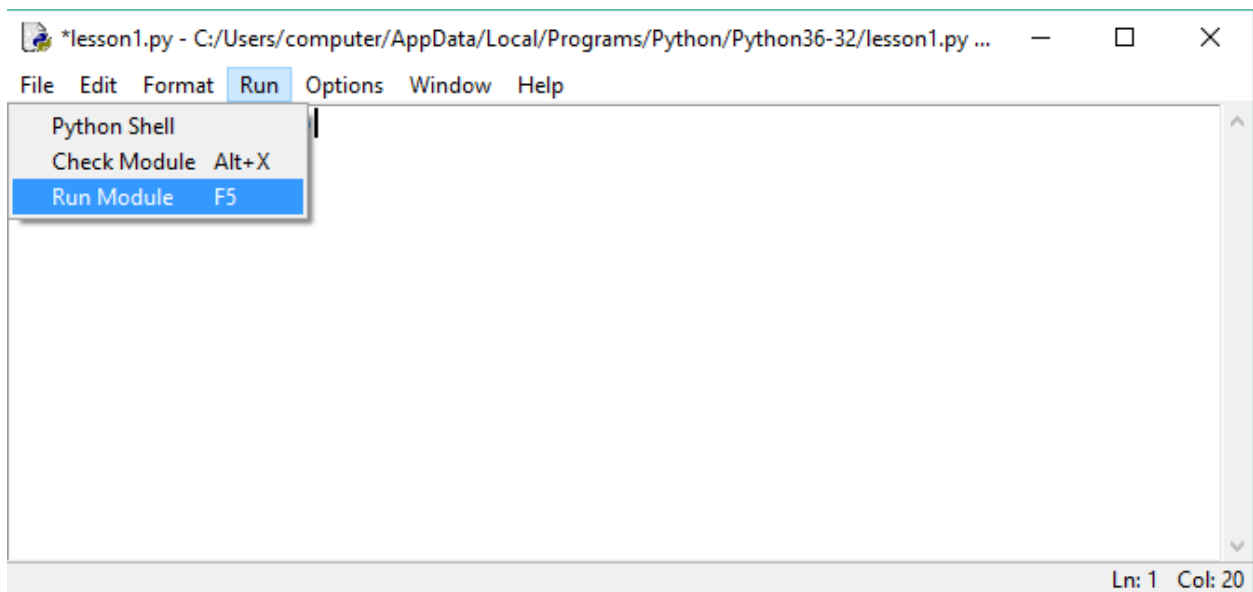


It's now time to write your first Python Programming Statement. Your program will print a welcome message to the computer screen. In the Python Editor type:

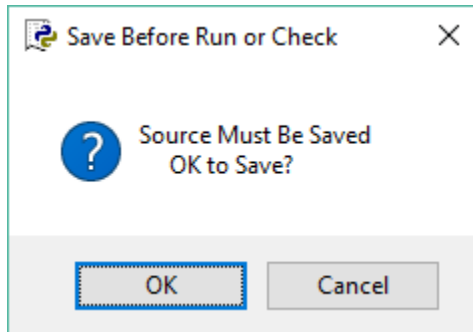
print("Welcome to my program")



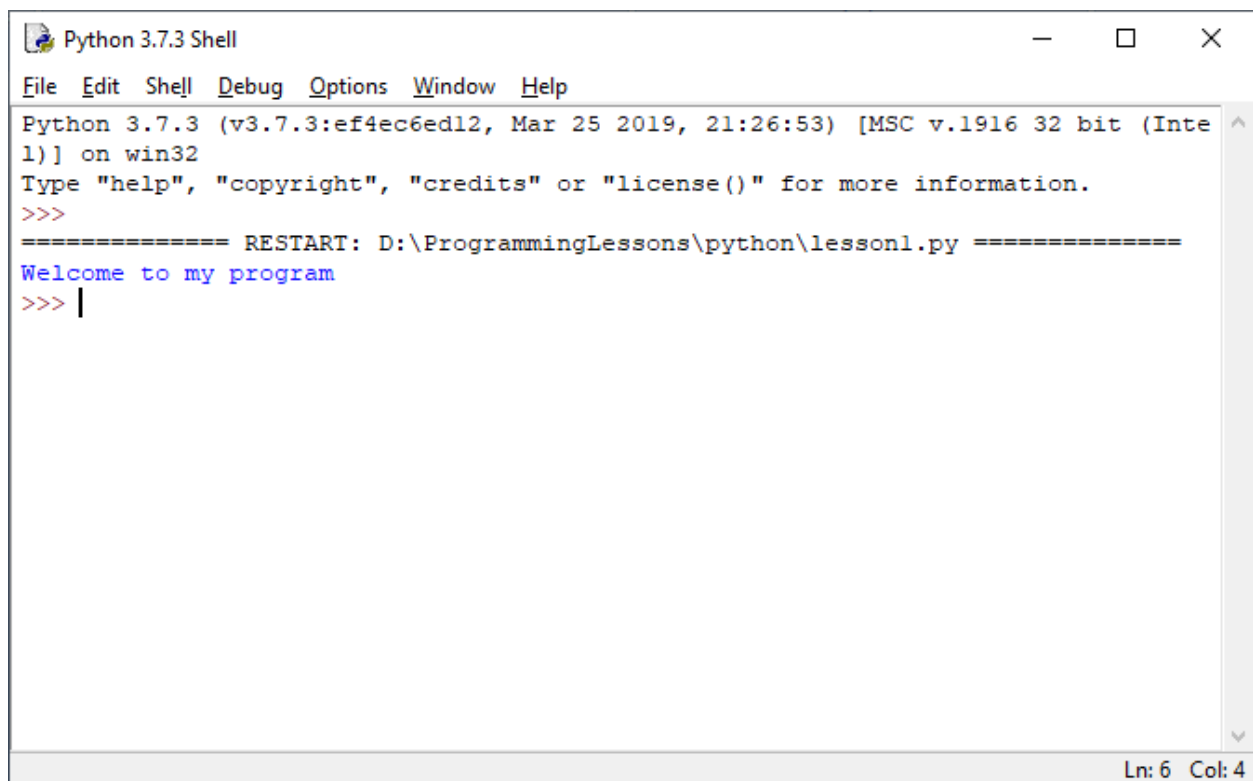
To run your program select Run Module from the Run Menu.



Select OK



“Welcome to my program” is now printed on the screen in the Python execution window. The **print** statement was used to print the “Welcome to my program” message on the screen .

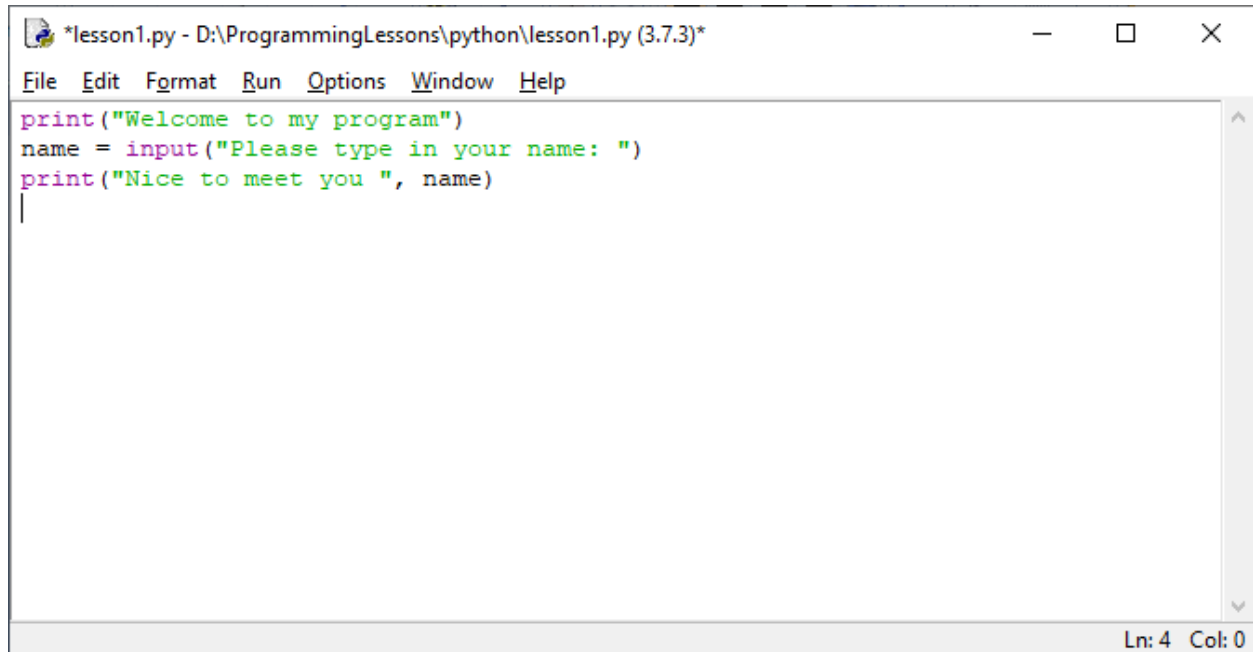


```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:\ProgrammingLessons\python\lesson1.py =====
Welcome to my program
>>> |
```

The next thing we need to do is get a value from the keyboard using an **input** statement. We will ask the user to type in their name and then greet them.

Type in the following statements in the Python editor right after the Hello World statement:

```
name = input("Please type in your name: ")  
print("Nice to meet you ", name)
```

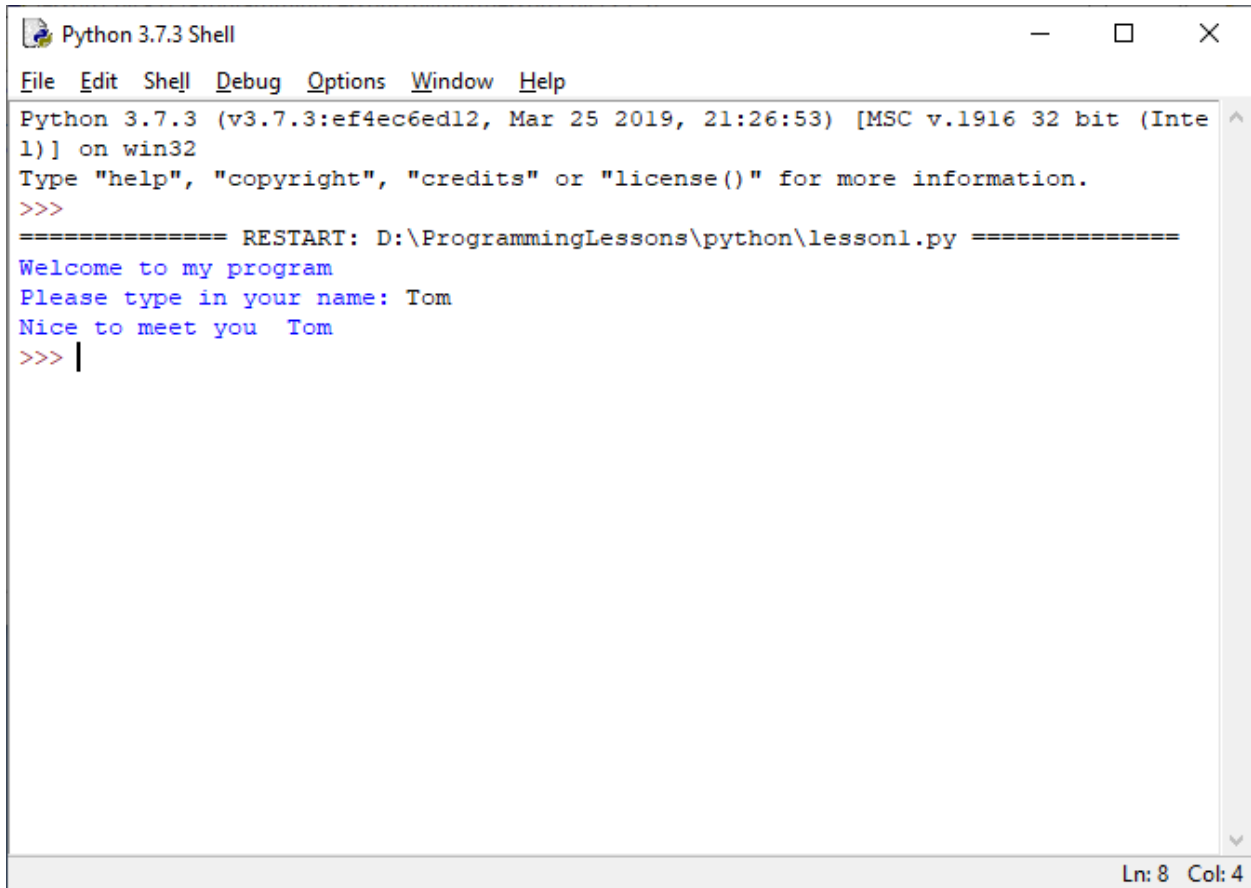
A screenshot of a Python IDE window titled '*lesson1.py - D:\ProgrammingLessons\python\lesson1.py (3.7.3)*'. The window has a menu bar with 'File', 'Edit', 'Format', 'Run', 'Options', 'Window', and 'Help'. The main text area contains the following code:

```
print("Welcome to my program")  
name = input("Please type in your name: ")  
print("Nice to meet you ", name)  
|
```

The status bar at the bottom right shows 'Ln: 4 Col: 0'.

Now run your program, you will get something like this:

(You may want to close the previous Python execution window before running.)



```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:\ProgrammingLessons\python\lesson1.py =====
Welcome to my program
Please type in your name: Tom
Nice to meet you Tom
>>> |
```

Ln: 8 Col: 4

Recapping: The **print** statement writes messages and values on the screen. The messages and values to be printed out are enclosed in round brackets and separated by commas. The **input** statement prompts the user with a text message and reads a value from the key board and stores the value in a **variable**. Variables store values. The **input** statement stores the name of the person in the variable name. Python has two types of values **string** values and **numeric** values. String values are messages enclosed in double or single quotes like "Hello World" or 'Hello World' where as numeric values are numbers like 5 and 10.5 Numeric values without decimal points like 5 are known as an **int** and numbers with decimal points like 10.5 are known as a **float**. Variables store string or numeric values that can be used later in your program. The variable **name** stores the string value entered from the keyboard, in this case the person's name.

```
name = input("Please type in your name: ")
```

The print statement prints out the string message "Nice to meet you" and the name of the user stored in the variable name.

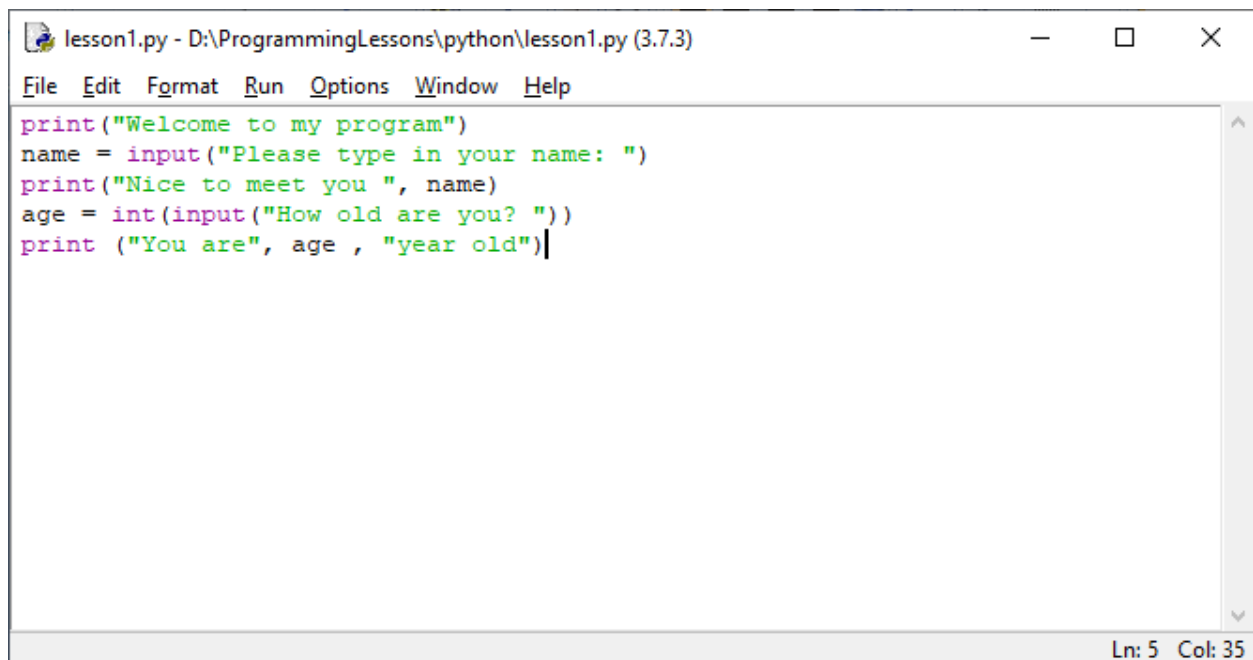
```
print("Nice to meet you ", name)
```

Note inside the print statement the string message and variable name are enclosed in round brackets and the values are separated by commas. Round brackets are important in Python. The opening round bracket '(' introduces the start of the values, the closing ')' round bracket species the end of the values.

We now ask the user how old they are.

Type in the following statements at the end of your program and then run the program.

```
age = int(input("How old are you? "))  
print ("You are", age , "years old")
```

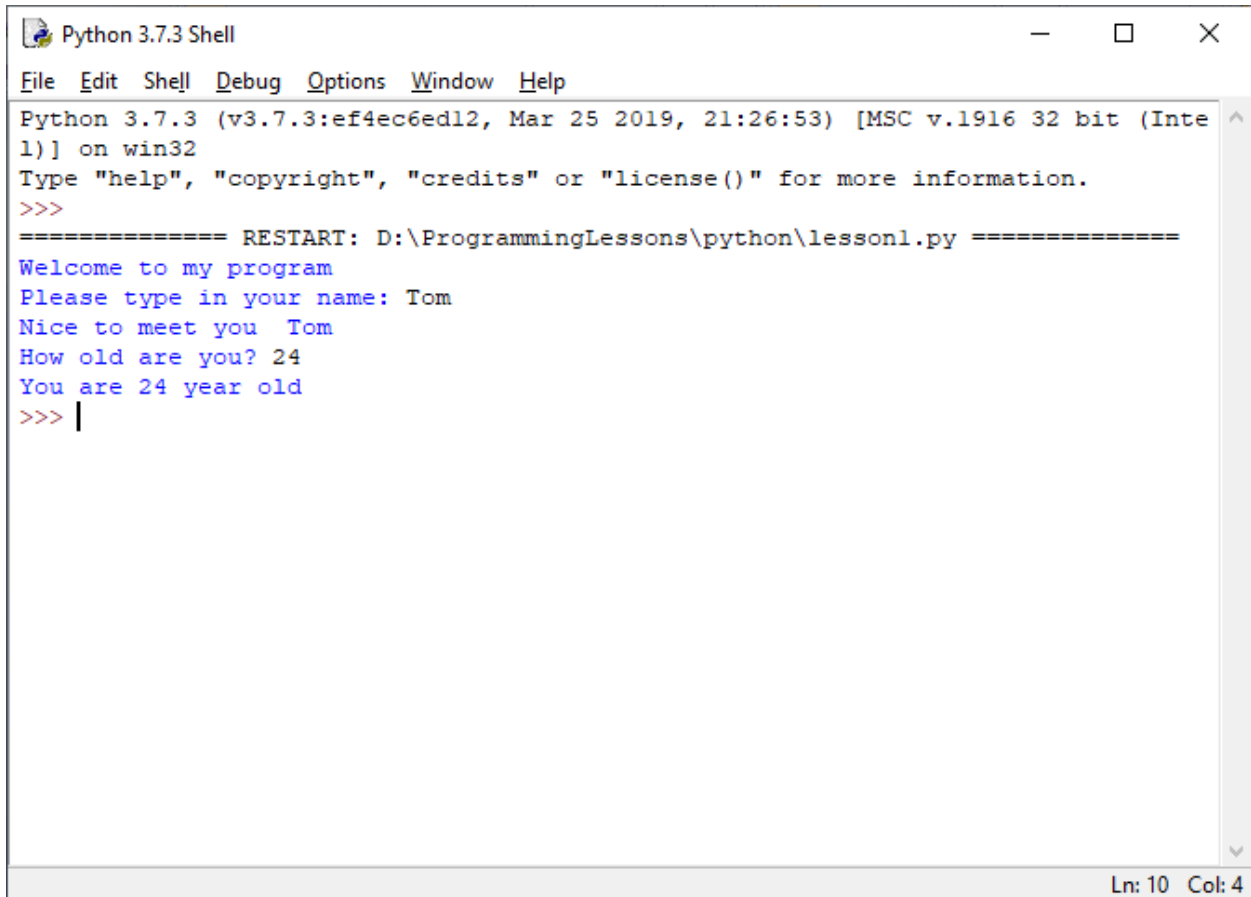


The screenshot shows a window titled "lesson1.py - D:\ProgrammingLessons\python\lesson1.py (3.7.3)". The window contains a menu bar with "File", "Edit", "Format", "Run", "Options", "Window", and "Help". The main area displays the following Python code:

```
print("Welcome to my program")  
name = input("Please type in your name: ")  
print("Nice to meet you ", name)  
age = int(input("How old are you? "))  
print ("You are", age , "year old")
```

The status bar at the bottom right indicates "Ln: 5 Col: 35".

Run the program and enter Tom for name and 24 for age you will get something like this:



```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:\ProgrammingLessons\python\lesson1.py =====
Welcome to my program
Please type in your name: Tom
Nice to meet you Tom
How old are you? 24
You are 24 year old
>>> |
```

Recapping: The **input** statement asks the user to enter their age. The **int** statement converts the entered age to a **int** number and the variable `age` holds the age value. We need to convert the string input to a numeric value using the **int** statement.

```
age = int ( input ("How old are you? "))
```

This is a 2 step process we first get the age of the person as a string using the `input` statement. The age at this point is a string value (digits are considered string letters when read by the `input` statement). The input value is converted to a number value using the `int` statement.

```
age = input ("How old are you? ")
```

```
age = int ( age)
```

Variables in Python can hold any value, string or number at any time.

The print statement is used to print out the messages, the person's name and age.

```
print (name,"You are", age , "years old")
```

If you have got this far then you will be a great python programmer soon.

Most people find Programming difficult to learn. The secret of learning program is to figure out what you need to do and then choose the right programming statement to use. If you want to store a value use a variable. If you want to print messages and values to the screen you use a **print** statement. If you want to get values from the keyboard, you use an **input** statement. If you need to input a numeric value, you use a **int** statement or **float** statement on the **input** statement to convert the input value to a numeric value.

Here is the complete program again:

```
name = input("Please type in your name: ")  
print("Nice to meet you ", name)  
age = int(input("How old are you? "))  
print ("You are", age , "years old")
```

Python data types:

With Python you do not need to specify what data type a variable is suppose to hold. Python figures this out for you automatically.

```
i = 5 # integer data type  
f = 10.5 # float data type  
d = 10.1234512 # double precision data type (exponential)  
c = 2 + 3j # complex data type  
b = True # boolean data type  
s = "Hello" String data type
```

One of the big draw backs of Python is you do not know what kind of data a variable represents. You can use `type(x)` to find out.

```
print(type(i)) # <class 'int'>
print(type(f)) # <class 'float'>
print(type(d)) # <class 'float'>
print(type(c)) # <class 'complex'>
print(type(b)) # <class 'bool'>
print(type(s)) # <class 'str'>
```

Introduction to Functions

Functions allow you to group many programming statements together so that you can reuse them repeatedly in your Python program. The most common function to use is the main function. We will now group our previous programming statements in a main function and then call the main function from the python script. Type in the following Python program, use tabs or spaces for the indentation. Indentation is very important on python programs. You must use proper indentation.

```
def main():
```

```
    print("Welcome to my program")
    name = input("Please type in your name: ")
    print("Nice to meet you ", name)
    age = int(input("How old are you? "))
    print ("You are", age , "years old")
```

```
main()
```

Now run your program, it will do the same thing as the previous program.

All functions in Python start with the key word **def** which means define function. Function name's end with 2 rounds brackets (). The round brackets distinguish a function name from a variable name. A function may receive values that are enclosed within the round brackets. Think that the round brackets are a portal mechanism for the function to receive values.

After the function name and the round brackets () there is a colon : the colon states there are programming statements to follow that belong to the current function name. Programming statements in a function are indented with a tab or spaces. All indented statements following **def** and the function name belong to the function. In the above function our function name is **main**. This indentation makes python very awkward to use and is the most cause of many program errors and unexpected program executions. Fortunately, you will get use to indentation and realize its importance in a python program to define the python program structure.

The last program statement **main()** is un-indented indicating the end of the main function and the start of a new programming statement. The main programming statement calls our main function. It has the same name as the main function and includes round brackets to indicate a function call.

If you get your program running then you are doing good.

Python has many built in functions that you can use, that make python programming easier to use. You already used some of them **print**, **input**, **int** and **float**. As we proceed with these lessons you will learn and use many more functions.

Lesson 1 Skill testing questions

1. What is a program?
2. What is a variable?
3. What is a data type, name some data types
4. What does a programming statement do?
5. What does the print statement do?
6. What does the input statement do?
7. What is a function?
8. What is the purpose of the main function?
9. How do you call the main function

Lesson 1 Homework:

Write a python program that ask someone what their profession **title** is, like Doctor, Lawyer, Salesman etc. and what their annual salary is. Next print out their profession **title** and how much money they make. You can use the **float** function to convert an input value into a number

```
salary = float(input("How much money do you make? "))
```

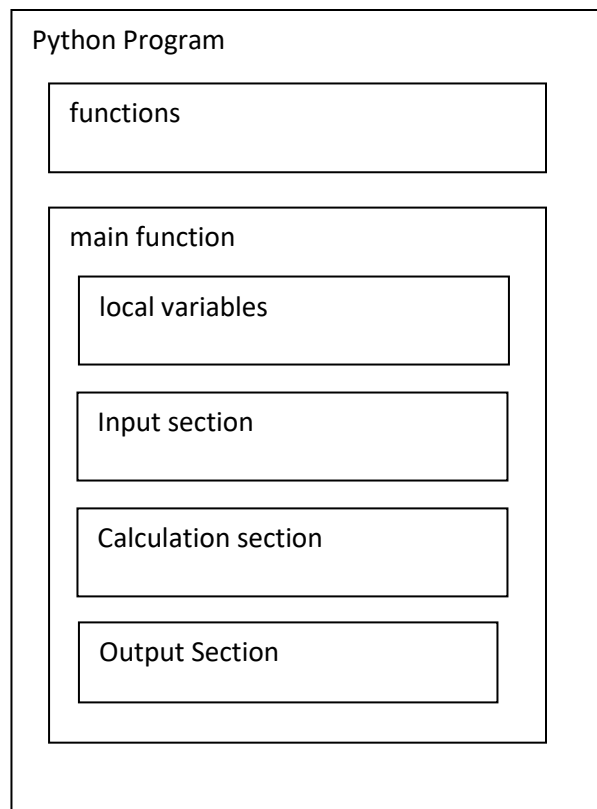
Use a main function to call your programming statements.

Your program would run like this:

```
Welcome to my Program
What is your profession title? Doctor
How much money do you make? 120000
You are a Doctor
You make 120000 dollars
```

Call your Python homework program, homework1.py

Python Program Format



LESSON 2 Functions

Functions allow your program to be more organized and allow programming statements to be re-used so duplication is avoided. We will make a **welcome**, **enterName** ad **enterAge** functions.

Functions usually are defined at the top of the program in order as they are used. The main function is the last one because it will call all the proceeding functions. When a function is called in a programming statement it means it is executed. In a Python script the functions must be defined before they are used.

Here is our program now divided into functions. Type in the following program and save as Lesson2.py.

```
def welcome():
```

```
    print("Welcome to my program")
```

```
def enterName():
```

```
    name = input("Please type in your name: ")
```

```
    return name
```

```
def enterAge():
```

```
    age = int(input("How old are you? "))
```

```
    return age
```

```
def displayInfo(name, age):
```

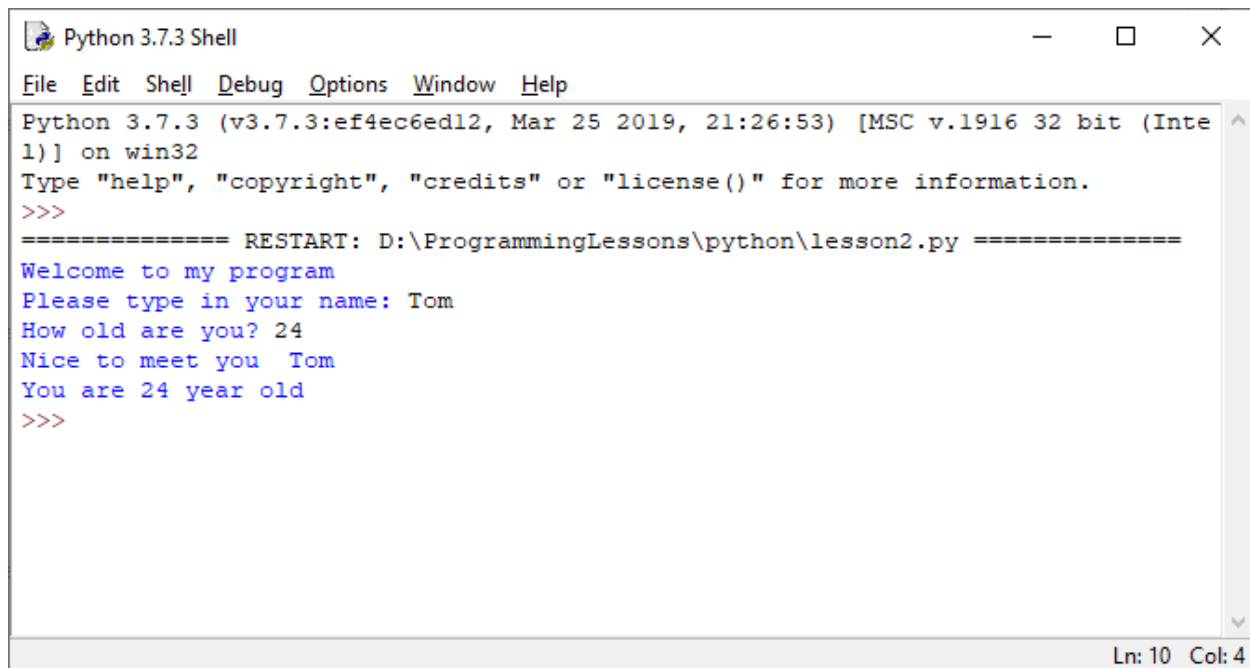
```
    print("Nice to meet you ", name)
```

```
    print ("You are", age , "years old")
```



```
def main():  
    welcome()  
    name = enterName()  
    age = enterAge()  
    displayInfo(name, age)  
  
main()
```

Run the program and you will get the same results as in program lesson1.py.



```
Python 3.7.3 Shell  
File Edit Shell Debug Options Window Help  
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>>  
===== RESTART: D:\ProgrammingLessons\python\lesson2.py =====  
Welcome to my program  
Please type in your name: Tom  
How old are you? 24  
Nice to meet you Tom  
You are 24 year old  
>>>  
Ln: 10 Col: 4
```

Functions make your program more organized and manageable to use. Functions have three different purposes. Functions can receive values, execute programming statements and return values. Functions receive values inside the round brackets following the function name. The values are separated by commas. The welcome function just prints a statement and receives no values or returns no value.

```
def welcome():  
    print("Hello World")
```

The getName function gets a name from the keyboard and return the name value using the return statement.

```
def enterName():  
    name = input("Please type in your name: ")  
    return name
```

The getAge function gets an age value from the keyboard and returns an age value using the return statement. Note we use the **int** function to convert the string value to a numeric value.

```
def enterAge():  
    age = int(input("How old are you? "))  
    return age
```

The p display function receives a name and age value to print out, but return's no value. **def displayInfo(name, age):**

```
    print("Nice to meet you ", name)  
    print ("You are", age , "years old")
```

The **name** and **age** inside the round brackets of the **displayInfo** function definition statement are known as **parameters** and contain values to be used by the function. The parameters just store values from the **calling** function and are not the same variables that are in the calling function. The calling function is the function that calls another function. Although the parameter names and values may be same as in the calling function variable names, they are different memory locations. The main purpose of the parameters is to transfer or pass values to the function.

The main functions call the preceding functions to run them and store the values in variables and pass the stored variable values to the other functions. Calling a function means to execute the function. The values that are passed to the called function from the calling function are known as **arguments**.

Variables inside a function are known as **local variables** and are known to that function only. Name and age are local variables in the main function but are also arguments to the **displayInfo** function.

```
def main():  
    welcome()  
    name = enterName()  
    age = enterAge()  
    displayInfo(name, age)  
  
main()
```

Notice, our main function is much smaller and our program is more organized. It's now time to comment your program. All programs need to be commented so that the user knows what the program is about. Just by reading the comments in your program you will know exactly what the program is supposed to do. We have two types of comments in python. Header comments, that are at the start of a program or a function. They start with 3 double quotes and end with three double quotes and can span multiple lines like this.

```
"""  
Program to read a name and age from a user and  
print the details on the screen  
"""
```

Other comments are for one line only and explain what the current or proceeding program statement it is to do,

The one-line comment starts with a # like this:

```
# function to read a name from the key board and return the value
```

We now comment the program as follow. Please add all these comments to your program.

```
"""
```

```
Program to read a name and age from a user and print  
the details on the screen
```

```
"""
```

```
# function to print a welcome message
```

```
def welcome():
```

```
    print("Welcome to my program")
```

```
# function to read a name from the key board are return the value
```

```
def enterName():
```

```
    name = input("Please type in your name: ")
```

```
    return name
```

```
# function to read an age from the key board are return the value
```

```
def enterAge():
```

```
    age = int(input("How old are you? "))
```

```
    return age
```

```
# function to print out a person's name and age
```

```
def displayInfo(name, age):
```

```
    print("Nice to meet you ", name)
```

```
    print ("You are", age , "years old")
```

main function to run program

def main():

welcome() # welcome user

name = enterName() # get user name

age = enterAge() # get user age

displayInfo(name, age) # print user name an age

main() # call main function to run program

Lesson 2 Homework:

Take your homework program from Lesson1.py that stored the title and salary of a Profession. Make functions welcome, enterTitle, enterSalary, displayInfo and main.

Use the functions to print a welcome message, get a profession title, get a salary and print out the professions details.

You can call your python homework 2 program, homework2.py

Lesson 2 Skill testing questions

1. What is the purpose of a function?
2. How do you define a function
3. How do you call a function
4. What is the purpose of the round brackets when you define a function
5. What is the purpose of the round brackets when you call a function
6. What is the difference of defining a function and calling a function
7. Why do we need functions?
8. How do you return values from a function?

LESSON 3 CLASSES

We now take a big step in Python Programming. This is very important step to take. **Classes** represent another level in program organization. They represent programming units that contain variables to store values and contain functions to do operations (calculations) on these variable. This concept is known as **Object Oriented Programming**, and is a very powerful concept. It allows these programming units to be used over again in other programs. The main benefit of a class is to store values and do operations on them transparent from the user of the class. It is very convenient for the programmers to use classes. They are like building blocks that allow one to create many sophisticated programs with little effort.

A class starts with the keyword **class** and the class name like this:

```
class Person:
```

The class uses another keyword **self** that indicates which variables and functions belong to this class. The keyword **self** is a little awkward to use, but we have no choice but to accept and use properly. Class definitions are little more automatic in other programming languages. Classes in Python are just probably an add on hack. All functions in a Python class must contain the **self** keyword.

We now convert our previous program to use a class. We will have a Person class that has variables to stores a name and age of a person and have functions to do operations on them, like initializing, retrieval, assignment and output. Type in the following class into a python file called lesson3.py

```
"""
Person Class to store a person's name and age
"""

# define a class Person
class Person:

    # initialize Person
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # return name
    def getName(self):
        return self.name

    # return age
    def getAge(self):
        return self.age

    # assign name
    def setName(self,name):
        self.name = name

    # assign age
    def setAge(self, age):
        self.age = age
```

```
# return person info as a string
def __str__(self):
    s = "Nice to meet you " + self.name + "\n"
    s += "You are " + str(self.age) + " years old"
    return s
```

recapping:

The Person class definition starts with the class key word and class name Person

class Person:

A class contains an **__init__()** function that initializes the class. This **__init__()** function is also known as a **constructor**. (**__** is 2 under scores) The mechanism that allocates memory in the computer for the variables defined in the class, is known as **instantiation**. When a class is instantiated it is known as an **object**. A class refers the class definition code that is typed into the program, where as an object refers to the memory that is allocated for the values of the variables defined in the class.

initialize Person

```
def __init__(self, name, age):
    self.name = name
    self.age = age
```

Notice the **self** keyword in the constructor parameter list. The **self** keyword is also passed to every function in the Person class and represents the memory location of the Person class variable values. The programming statements inside the constructor define the variables name and age belonging to the Person class. Name and age are assigned values from the parameters name and age.

```
self.name = name
self.age = age
```


The keyword **self** specifies which variables belongs to the Person class. The parameter name and age are just used to pass values to be assigned to the variables of the Person class and are not the same ones in the Person class.

The **self** key word also distinguished which variables belong to the class and which variables are local variables or a parameter.

The get functions also known as **getters** and just return values of the variables stored in the Person class. Again, you notice the self keyword.

```
# return name  
  
def getName(self):  
    return self.name
```

```
# return age  
  
def getAge(self):  
    return self.age
```

We also have set functions known as **setters** that allow the user of the class to assign new values to the variable belonging to the Person class.

```
# assign name  
  
def setName(self,name):  
    self.name = name
```

```
# assign age  
  
def setAge(self, age):  
    self.age = age
```

You will notice the parameter list has the keyword `self` and an additional parameter to assign the name or age value. Again, the `self` keyword distinguishes the person variables from the parameters since they both have the same names.

All classes should have a `__str__()` function so that it can easily return the class variables as a string message.

```
# return person info as a string
```

```
def __str__(self):
```

```
    s = "Nice to meet you " + self.name + "\n";
```

```
    s += self.name + " You are " + str(self.age) + " years old"
```

```
    return s
```

Notice we have no print statement in our `__str__` function. We assign information to the local variable `s` and return the `s` value. A local variable is just known to the function it resides in. The `s` variable uses the `+` operator to join values together as a message. Unfortunately, the `+` operator only joins string variables in case of numeric value. In this case a `str` function is used to convert a numeric value to string value. The `str` function is a common built in python function that return a string value.

The class definition should not contain any input or output statements. A class must be a reusable program unit, not dependent on any input or output print statements. The purpose of the class is to contain information that can be easily accessed.

Therefore, additional functions not belonging to the Person class must provide all the input and output statements.

When you use a class definition in a program it becomes an object. A object is simply allocated memory for the variables defined in the class definition. A class is considered a user data type. Objects are made from class definitions. Just like a house is made from a set of drawing plans. The house is the object and the drawing plans is the class definition.

We make a Person object like this:

```
p = Person('Tom',24)
```

When you make a object from a class definition it is known as **instantiating**. When a class definition is instantiated computer memory is allocated for the variables defined in the class. The variable **p** holds the location of the computer memory where the Person object was created. The p becomes the value for the self keyword defined in the Person class definition. The self keyword in the class needs to know which object you are using.

We give the Person `__init__` function the person's name and the persons age.

```
def __init__(self, name, age):
```

```
    self.name = name
```

```
    self.age = age
```

We can also print out the person's name and age by calling the `__str__` function automatically using

```
    print(p)
```

This would be the same thing as writing `print(p.__str__())` or `print(str(p))`

Now put the above statements in a main function and run the program.

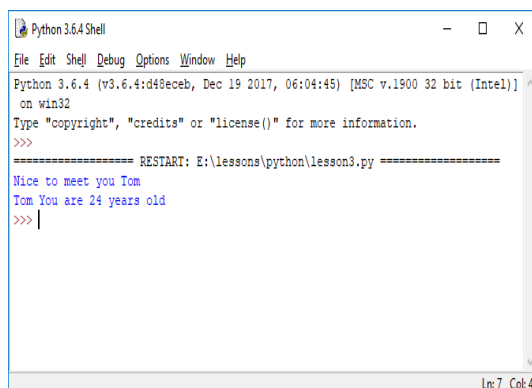
```
def main():
```

```
    p = Person('Tom',24)
```

```
    print(p)
```

```
main()
```

You should get something like this



```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: E:\lessons\python\lesson3.py =====
Nice to meet you Tom
Tom You are 24 years old
>>> |
```

We can expand our program to get a person's name and age from the keyboard. We will use the input and output functions from our previous program.

Type in or copy the following python statements from lesson2.py and put at the bottom of your lesson3.py file and also add the additional line to the main function as follows.

```
# function to print a welcome message  
def welcome():  
    print("Hello World")  
  
# function to read a name from the key board and return the value  
def enterName():  
    name = input("Please type in your name: ")  
    return name  
  
# function to read an age from the key board and return the value  
def enterAge():  
    age = int(input("How old are you? "))  
    return age  
  
# main function to instantiate class Person and run program  
def main():  
    welcome() # welcome user  
    name = enterName() # get user name from keyboard  
    age = enterAge() # get user age from key board  
    p = Person(name, age) # create Person class  
    print(p) # print user name and age from person class  
  
main() # call main function
```

Notice we create the Person object from the person class definition using the following statement:

```
p = Person(name, age) # create Person object
```

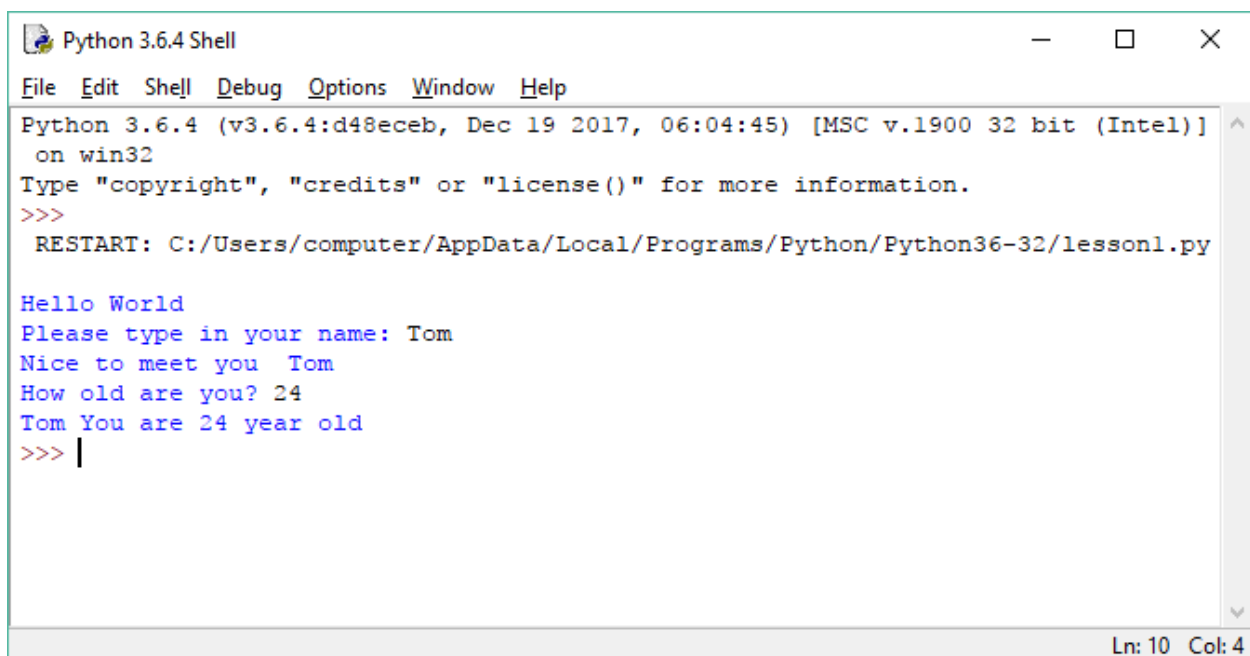
Objects are allocated memory in a computer for the variables defined in the class when the program runs. Objects are created from class definitions.

This statement calls the `__init__()` function of the person class to create the person object and initialize it with the values name and age.

Using the `p` variable the `print` statement automatically calls `__str__()` function

```
print(p)
```

After typing in the class and main function in a Python program `lesson3.py` and running the program, you will get the same output as the previous program.



```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:/Users/computer/AppData/Local/Programs/Python/Python36-32/lesson1.py

Hello World
Please type in your name: Tom
Nice to meet you Tom
How old are you? 24
Tom You are 24 year old
>>> |
```

Ln: 10 Col: 4

default parameters

Constructors and other functions can also have predefined parameter values. This comes in handy when you want to assign the values later or do not know what the values are supposed to be. You can make a default constructor like this.

```
def __init__(self, name="", age=0):
```

```
    self.name = name
```

```
    self.age = age
```

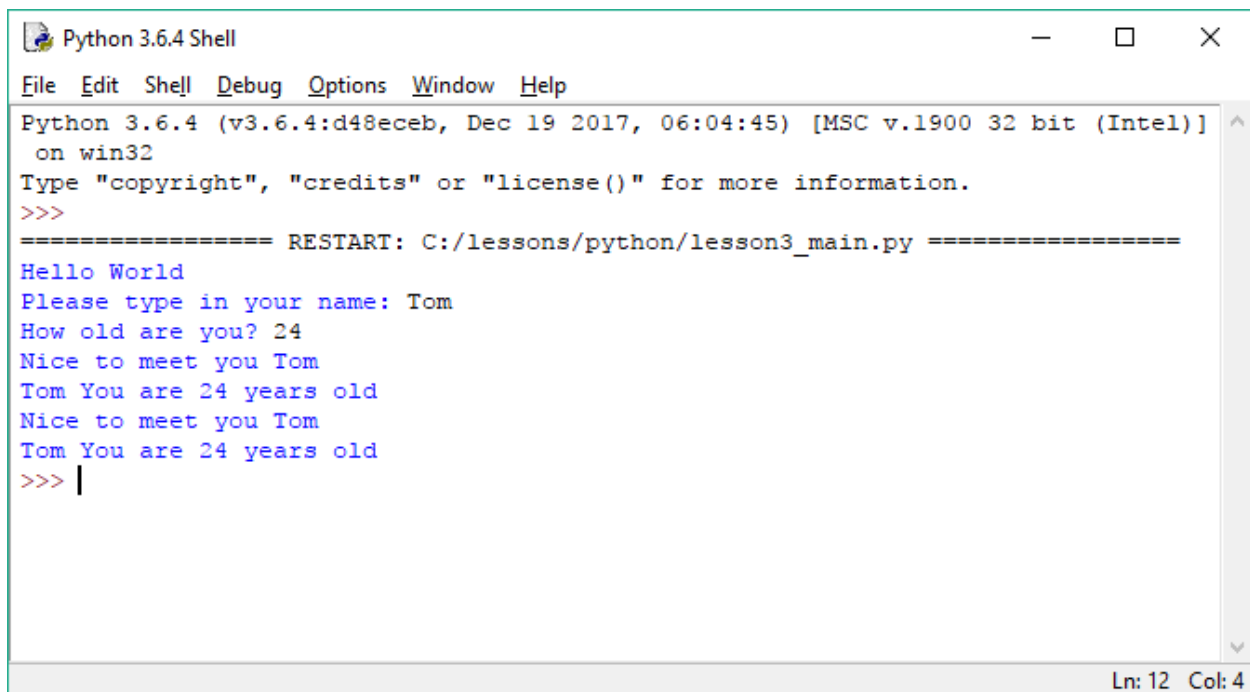
Notice the parameter name is pre-initialized to "" and the parameter age is pre-initialized to 0.

to do:

Change your Person class to use a default constructor. Make a default person called p2 like this:

```
p2 = new Person() # create default Person object
```

Use the getters from the **p** person object and assigned the values to the p2 person object using the setters, then print out the p2 details. You should get something like this:



```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/lessons/python/lesson3_main.py =====
Hello World
Please type in your name: Tom
How old are you? 24
Nice to meet you Tom
Tom You are 24 years old
Nice to meet you Tom
Tom You are 24 years old
>>> |
```

If you can do this, you are almost great python programmer!

Lesson 3 Homework Part 1

Take your homework program from Lesson 2 and make a Profession class that stores a profession title like: Doctor, Lawyer, Salesman, Secretary etc. and a salary.

Make initializer `_init_` that receives a title and a salary.

Make getters and setters for title and salary.

Lastly make a `__str__` function use to print out the profession's title and salary.

In the main method make 2 Profession objects. One using the key board functions and the other one with hard coded values.

Print both objects on the screen.

Put all your homework 3 in a python file called Homework3.py. You will still need the standalone functions **welcome**, **enterTitle**, **enterSalary** from homework 2.

INHERITANCE

The beauty of classes is that they can be extended to increase their functionality. We can make a Student class to use the variables and functions from the Person class. This is known as **inheritance**.

A Student class will have an additional variable called student that will represent a string student id number. Using inheritance, the student class will be able to use the variables and functions of the Person class. The Person class is known as the **super** or **base** class and the Student class is known as the **derived** class. The Person class knows nothing about the Student class where as the Student class knows all about the Person class.

Create a class called Student below the Person class using this statement

```
class Student (Person):
```

The above statement means to define a class Student that inherits the Person class.

Now make a constructor that will initialize the student name, age and student.

```
# initialize Student  
def __init__(self, name, age, studentid):  
    super().__init__(name, age)  
    self.studentid = studentid
```

Notice we pass the name and age to the super Person class by calling the Person.__init__() constructor and passing self, name and age

Note for python version 2.7 you need to Person class name instead of the super() function.

```
# initialize Student  
def __init__(self, name, age, student):  
    Person.__init__(self, name, age)  
    self.studentid = studentid
```

You should now be able to make the getID and setID getters and setters like this without our help.

```
# return student id  
def getId(self):  
    return self.studentid
```

```
# assign student id  
def setId(self, studentid):  
    self.studentid = studentid
```

The last thing you need to make the **__str__()** function. By using the **super** class name **Person** you can call functions directly from the super Person class inside the Student derived class. Here is the Student **__str__()** function


```

# return student info as a string
def __str__(self):

    s= super().__str__() + "\n"
    s += " having Student ID: " + self.studentid;
return s

```

Note: for python 2.7 you need to use the Person class name instead.

```

s = Person.__str__(self)

```

Once you got the Student class made then add programming statements to the lessons3_main.py file to obtain a student name, age and studentid. You will have to make an additional **enterID()** function to obtain a student id number from the key board.

```

# function to read a student id number from the key board
# and return the value
def enterIDnum():

    studentid = input("Please type in your student number: ")
return studentid

```

Next make a student object and use the obtained name, age and studentid then print out the student details.

```

s = Student(name, age, studentid) # create Student object
print(s)

```

You should get something like this:

```

Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/lessons/python/lesson3_main.py =====
Hello World
Please type in your name: Tom
How old are you? 24
Nice to meet you Tom
Tom You are 24 years old
Nice to meet you Tom
Tom You are 24 years old
Please type in your name: Sue
How old are you? 36
What is your Student ID? S1234
Nice to meet you Sue
Sue You are 36 years old having Student ID: S1234
>>> |

```

Derived class default parameters

The derived Student class can also have default parameters as follows.

```
# initialize Person
def __init__(self, name="", age=0):
    self.name = name
    self.age = age

# initialize Student
def __init__(self, name, age, idnum="S1234"):
    super().__init__(name, age)
    self.studentid = studentid
```

Using separate files for classes

Classes are usually put into their own files So put all the Person class code in a file called person.py. Put all the Student class code in a file called student.py.

On the top of the student.py file just below the header comment you need to tell the python interpreter to use the Person class

```
from person import Person
```

This means from the file person.py use the code from the Person class.

Put all the main functions in a file called lesson3_main.py. Lesson3_main.py needs some extra statements. We need to tell file lesson3_main.py to use the Person class and Student class. Put these statements just below the header comment of lesson3_main.py file near the top of the file.

```
from person import Person
```

This means from file person.py use the code from the Person class.

```
from student import Student
```

This means from file student.py use the code from the Student class.

Importing python pre-built modules

Python has lots of pre-built modules you can use like the **math** module

```
import math
print(math.pi) # 3.141592653589793
```

You can also give modules another name using the **as** directive.

```
import math as m
print(m.pi) # 3.141592653589793
```

Indicating the main module

You also need to tell the python interpreter that lesson3_main.py is the main file to run. You cannot run a class without a main program. Classes do not run by themselves. Put this statement just above the **main()** call statement at the bottom of the file.

```
if __name__ == "__main__":
    main() # call main function
```

Now run file lesson3_main.py and make sure everything still works.

LESSON 3 HOMEWORK Part 2

Make a class called Payroll derived from the Profession class where the professional get a designated bonus. In the Payroll class make constructor **__init__** that receives a title, salary and bonus. The Payroll **__init__** function should send the title and salary to the Profession **__init__** function. Make getter and setters for the bonus. Finally make a **__str__** function for the Payroll class. The Payroll **__str__** function should also call the **__str__** function from the Profession class.

The Payroll `_str_` function should print out the title, salary and bonus.

You will also need to make another standalone function `enterBonus`.

In the main method make 2 Payroll objects. One using the key board functions and the other one with hard coded values.

Put every thing in the same file `homework3.py` used in the previous homework.

Additional homework to do

Put the Profession class and Payroll classes in separate files called `profession.py` and `patrol.py` respectively.

```
from profession import Profession
```

```
from payroll import Payroll
```

In the main file call main using:

```
if __name__ == "__main__":  
    main() # call main function
```

You can still use the same file `homework3.py`

Lesson 3 Skill testing questions

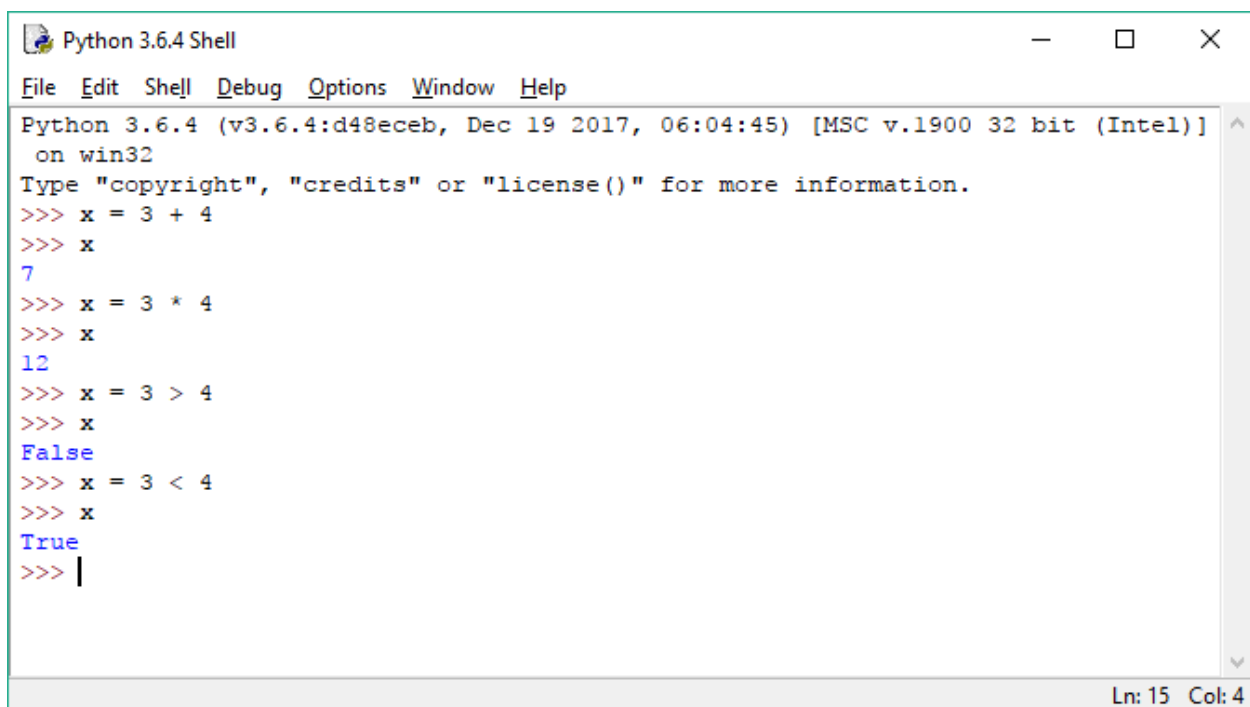
1. What is a class?
2. What is an object?
3. What is the difference between a class and an object
4. Why do we need classes?
5. What are the components of a class
6. What does the `__init__()` do?
7. What do getters do?
8. What do setters do?

9. What does the `__str__()` do?
10. What is inheritance?
11. How do you make a subclass?
12. What does `super()` do?

LESSON 4 OPERATORS

Operators

Operators do operations on variables like addition + , subtraction -, comparisons > (greater) < (less) etc. You can test Python programming statements directly on the Python shell, which is very convenient to use. You just type in the program statement into the python shell and it gets executed automatically. To see the value of a variable you just type in the variable name and the value is displayed automatically. Hint: To type in more than 1 line at a time end the last line with a extra enter to get back to the shell prompt.



```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> x = 3 + 4
>>> x
7
>>> x = 3 * 4
>>> x
12
>>> x = 3 > 4
>>> x
False
>>> x = 3 < 4
>>> x
True
>>> |
```

We now present all the Python operators. You can type all the examples in the python shell or put in a python file called lesson4.py

If you use a python file, then you will have to use print statements to see the result on the screen like this:

```
print (3 + 4) # would print 7
```

or like this using variables:

```
x = 5
y = 3
print (x, "+", y, "=", x + y) # would print 3 + 4 = 7
```

Arithmetic Operators

Arithmetic operators are used to do operations on numbers like addition and subtraction.

Operator	Description	Example	Result
+	Add two operands or unary plus	x = 3 + 2	5
-	Subtract right operand from the left or unary minus	x -= 3 - 2 x = -2	1 -2
*	Multiply two operands	x = 3 * 2	6
/	Divide left operand by the right one	x = 5 / 2	2.5
%	Modulus - remainder of the division of left operand by the right	x = 5 % 2	3
//	Floor division - division that results into whole number	x = 5 // 2	2
**	Exponent - left operand raised to the power of right	x = 5 ** 2	25

Comparison Operators

Comparison operators are used to compare values. It either returns **True** or **False** according to the condition. In python true and false start with capital letters **True** or **False**.

x = 5

y = 3

print (x,">",y, " = ",x > y) # would print 5 > 3 = False

Operator	Description	Example	Result
>	Greater than - True if left operand is greater than the right	5 > 3	True
<	Less than - True if left operand is less than the right	3 < 5	True
==	Equal to - True if both operands are equal	5 == 5	True
!=	Not equal to - True if operands are not equal	5 != 5	True
>=	Greater than or equal to - True if left operand is greater than or equal to the right	5 >= 3	True
<=	Less than or equal to - True if left operand is less than or equal to the right	5 <= 3	True

Logical Operators

Logical operators are the **and**, **or**, **not** boolean operators.

x = True

y = False

print (x,"and",y, " = ",x and y) # would print True and True = True

Operator	Description	Example	Result
And	True if both the operands are true	True and True	True
Or	True if either of the operands is true	True or False	True
Not	True if operand is false (complements the operand)	Not False	True

Compound Comparison Operations

You may also combine the Comparison operators with the Logical operators like to form compound comparisons:

```
5 > 3 and 3 != 6
```

```
3 < 5 or 3 == 6
```

```
x = 3
```

```
y = 5
```

```
print (x, ">", y, "and", x, "<", y, " = ", x > y and x < y) # 3 > 5 and 3 < 5 = False
```

Binary Numbers

All numbers in a computer are stored as binary numbers. Binary numbers (base 2) just has 2 digits 0 and 1 whereas decimal numbers have 10 digits 0 to 9. We also have hexadecimal (base 16) numbers 0 to F that represent decimal numbers 0 to 15. We use the letters A to F to represent decimal numbers 10 to 15.

Here are the binary and hexadecimal numbers for decimal numbers 0 to 15.

Decimal	Binary	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8

9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Bitwise Operators

Bitwise operators act on operands as if they were binary digits. It operates bit by bit. Binary numbers are base 2 and contain only 0 and 1's. Every decimal number has a binary equivalent. Every binary number has a decimal equivalent. For example, decimal 2 is 0010 in binary and decimal 7 is binary 0111.

`print(x, "|"n,y, "=", 10 | 4); // would print out 10 | 4 = 7`

In the table below: Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

Operator	Description	Example and Result
&	Bitwise AND 1 if both operands are 1 otherwise 0	x & y = 0 (0000 0000)
	Bitwise OR 1 if either operands are 1	x y = 14 (0000 1110)
^	Bitwise XOR same values are 0 opposite vales are 1	x ^ y = 14 (0000 1110)
~	Bitwise NOT reverse bits	~x = -11 (1111 0101)

You can use the **bin** function to print out numbers in binary.

```
print(bin(5))
```

```
'0b0101'
```

You may want to use variables instead like this:

```
x = 0
```

```
y = 1
```

```
print(x & y)
```

using 0 and 1's rather than numbers make the bitwise operations easier to understand

and &	or	xor ^
0 & 0 = 0	0 0 = 0	0 ^ 0 = 0
0 & 1 = 0	0 1 = 1	0 ^ 1 = 1
1 & 0 = 0	1 0 = 1	1 ^ 0 = 1
1 & 1 = 1	1 1 = 1	1 ^ 1 = 0

The ~ operator reverse the bits. 0 becomes 1 and 1 becomes -

```
10 = 0000 1010
```

```
~ 1111 0101
```

Negative binary numbers have a 1 at the start known as the msb (most significant bit)

1111 0101 is actually =-11

You can use 2's complement to convert a positive binary number to a negative binary number or a negative binary number to a positive binary number.

	0000 1011	1111 0101
Step 1 complement binary number	1111 0100	0000 1010
Step 2 add	1	1
	-----	-----
	1111 0101 (-11)	1111 1011 (11)

Shift Operators

Shift operators are used to multiply or divide numbers by powers of 2

Multiply by powers of 2 shift bit right

$x \ll 1$ means to multiply x by 2^1 which is $x * 2$
 $x \ll 2$ means to multiply x by 2^2 which is $x * 4$
 $x \ll 3$ means to multiply x by 2^3 which is $x * 8$

Divide by powers of 2 shift bits left

$x \gg 1$ means to divide x by 2^1 which is $x / 2$
 $x \gg 2$ means to divide x by 2^2 which is $x / 4$
 $x \gg 3$ means to divide x by 2^3 which is $x / 8$

x = 2

y = 3

print(x,"<<", y,x << y) // 2 << 3 = 16

x = x << y

print(x,">>", y,x << y) // 16 >> 3 = 2

<<	left shift multiply by powers of 2	x = 2 x = x<< 3 2* 8 = 16
>>	right shift divide by powers of 2	x = x>> 2 x / 4 = 5

Assignment Operators

Assignment operators are used in Python to assign values to variables.

$x = 5$ is a simple assignment operator that assigns the value 5 on the right to the variable x on the left.

There are various compound operators in Python like `x += 5` that adds to the variable and later assigns the same. It is equivalent to `x = x + 5`.

Try them all!

```
x = 5
print(x)
x+=5
print("x += 5 = ",x); // x += 5 = 10
```

Operator	Compound	Equivalent
=	<code>x = 5</code>	<code>x = 5</code>
+=	<code>x += 5</code>	<code>x = x + 5</code>
-=	<code>x -= 5</code>	<code>x = x - 5</code>
*=	<code>x *= 5</code>	<code>x = x * 5</code>
/=	<code>x /= 5</code>	<code>x = x / 5</code>
%=	<code>x %= 5</code>	<code>x = x % 5</code>
**=	<code>x **= 5</code>	<code>x = x ** 5</code>
//=	<code>x //= 5</code>	<code>x = x // 5</code>
&=	<code>x &= 5</code>	<code>x = x & 5</code>
=	<code>x = 5</code>	<code>x = x 5</code>
^=	<code>x ^= 5</code>	<code>x = x ^ 5</code>
<<=	<code>x <<= 5</code>	<code>x = x << 5</code>
>>=	<code>x >>= 5</code>	<code>x = x >> 5</code>

Identity Operators

is and **is not** are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

```

x = 5
y = 5
print(x , " is ", x,x is y) # 5 is 5 = True
print(x , " is ", y,x is y) # 5 is 5 = False

```

Operator	Description	Example
is	True if the operands are identical	x is y
is not	True if the operands are not identical	x is not y

in Operator

The **in** operator test if a value is stored in a collection like a string

```

x = 'a'
y = 'apple'
print(x , " in ", y,x in y) # a in apple True
print(x , " not in ", y,x not in y) # a not in apple False

```

Operator	Description	Example
In	True if value in a collection	' a ' in 'apple '
not in	True if the value not in a collection	'a 'not in 'apple '

You should type in all the examples and try them out. You will be using them soon in the next lesson.

String Operators

String operators are used to do operations on strings like joining strings or getting parts of a string. You will be using string operators a lot.

```
# join two strings together
```

```
s1 = "hello"
```

```
s2 = "there"
```

```
s3 = s1 + s2
```

```
print(s3) # hellothere
```

```
# length of string
```

```
length = len(s3)
```

```
print(length) # 10
```

```
# get a character from string using an [index]
```

```
# (index start at 0)
```

```
c = s3[0]
```

```
print (c) # h
```

```
# get last character
```

```
c = s3[-1]
```

```
print (c) # e
```

You cannot assign a value to a string index, you will get a error message

```
s3[0] = 'x'
```

Strings are immutable meaning you cannot change them.

```
# get a substring using slices
```

```
# [start index=0 : end index=len()-1 : (optional step=1)]
```

```
# (defaults are start, end index and step by 1)
```

```
# print out character indexes 0 to 4
```

```
s4 = s3[0:5]
```

```
print (s4) # hello
```

print out character indexes 0 to 4 using default start index

```
s4 = s3[:5]
```

```
print (s4) # hello
```

print out character indexes 0 to 4 using default end index

```
s4 = s3[0:]
```

```
print (s4) # hello
```

print out all characters using default index's

```
s4 = s3[:]
```

```
print (s4) # hello
```

print first to last -1 letters

```
s4 = s3[0:-1]
```

```
print (s4) # hell
```

print first to last -1 letters using default start index

```
s4 = s3[:-1]
```

```
print (s4) # hell
```

print second to last -1 letters

```
s4 = s3[2:-1]
```

```
print (s4) # ll
```

print second to fourth letters

```
s4 = s3[2:4]
```

```
print (s4) # ll
```

reverse a string

```
s6 = s5[::-1]
```

```
print (s6) # erehtXolleh
```

reverse a range

```
s6 = s5[10:2:-1]
```

```
print (s6) # ehtXoll
```

add a character to middle of a string

```
s5 = s3[:5] + 'X' + s3[5:]
```

```
print (s5) # helloXthere
```



```
# replace a character in a string
s5 = s5[:5] + ' ' + s5[6:]
print (s5) # hellothere

# make string upper case
s8 = s6.upper()
print(s8) # EREHTXOLLEH

# make string lower case
s7 = s6.lower()
print(s7) # erehtxolleh

# change a character to a ASCII number
x = ord('A')
print(x) #65

#change a ASCII number to a character
c = chr(x)|
print(c) # A

#repeat a string using the * operator
s8 = 'Happy' * 2
print(s8) # 'HappyHappy'

# check if a character or substring is in a string
s = 'happy'
x = 'a' in s
print(x) # True
x = 'g' in s
print(x) # False

# returns the index of first occurrence of the substring.
# If not found, it returns -1.
s = 'happy'
i = s.find('a')
print(i) # 1
```

```
i = s.find('g')
print(i) # -1
```

note: find has optional start at stop arguments

```
i = s.find('a',1,5) # look for a starting at index 1 to index 5
```

returns the index of a substring inside the string.

If the substring is not found, it raises an exception.

try:

```
s = 'happy'
i = s.index('a')
print(i) # 1
```

except ValueError:

```
print('a not found')
```

note: find has optional start at stop arguments

```
i = s.index('a',1,5) # look for a starting at index 1 to index 5
```

format a string

method 1 using %

```
s9 = '$%.2f' % (10.4564)
```

```
print(s9) # 10.5
```

#method 2 using format

```
S9 = '${:.2f}'.format(10.4564)
```

```
print(s9) # 10.5
```

method 3 using f' formatter

```
x = 10.4564
```

```
s9 = f'${x:.2f}'
```

```
print(s9) # 10.5
```

Lesson 4 Homework

1. Print out if a number is even, using just use a print statement and an arithmetic operator
 2. Print out of a number is odd, using just use a print statement and an arithmetic operator
 3. Swap 2 numbers using a temporary variable, print out numbers before and after swapping them.
 4. In a print statement, add 2 numbers together and check if they are less than multiplying them together
 5. In a print statement, add 2 numbers together and check if they are less than multiplying them together and greater then multiplying them together.
 6. In a print statement, add 2 numbers together and check if they are less than multiplying them together or greater then multiplying them together.
 7. Multiply a number by 8 using a shift operator, print out numbers before and after shifting them.
 8. Divide a number by 8 using a shift operator, print out numbers before and after shifting them
 9. Make a string and replace the first letter with another letter
Example: change 'hello' to 'jello'
 10. Make a string and replace the last letter with another letter
Example: change 'jello' to 'jelly'
 11. Make a string and replace the middle letter with another letter
Example: change 'jelly' to 'jexly'
 12. Make a string. Split it in the middle, swap both parts and reverse the first part and change to upper case.
Example Change: 'jexly' change to: 'YLjex'
 13. Make a string. replace the last letter with the first letter.
Example Change: 'YLjex' change to: 'xLjeY'
- Call your python file homework4.py

LESSON 5 LISTS, SETS, TUPLES AND DICTIONARIES

Lists

Lists store many sequential values together. Lists are analogous to arrays in other programming languages. Lists in python are very powerful and can do many things. We now present many list examples. Put the following programming statements in a python file called lesson5.py and run it.

```
# To create an empty list
```

```
list1 = []  
print(list1)          # []
```

```
# To create a list with pre-initialized values
```

```
list2 = [1,2,3,4, 5]  
print(list2)          # [1, 2, 3, 4, 5]
```

```
# get the number of elements in an list
```

```
x = len(list2)  
print (x)             # 5
```

```
# Add a value to a list
```

```
list1.append(5)  
print(list1)          # [5]
```

```
# Get a value from a list at a specified location
```

```
x = list2[0]  
print(x)              # 1
```

```
# Get part of a slice of a list
```

```
# [start index=0 : end index=len()-1 : (optional step=1)]  
# (defaults are start, end index and step)  
# [:] would be the whole list using defaults  
list3 = list2[1:3]  
print(list3)          # [2, 3]
```

```

# get last value of list
list3 = list2[-1:]
print(list3)          # [5]

# get last 2 values of list
list3 = list2[-2:]
print(list3)          # [4, 5]

# get every other value
list3 = list2[0:5:2]
print(list3)          # [1,3,5]

# get all values except last value
list3 = list2[:-1]
print(list3)          # [1, 2, 3, 4]

# get all values except last 2 values
list3 = list2[:-2]
print(list3)          # [1, 2, 3]

# get all values except last 2 values
list3 = list2[-5:-2]
print(list3)          # [1, 2, 3]

# get every other values except last value
list3 = list2[-5:-1:2]
print(list3)          # [1, 3]

# reverse a list
list3 = list2[::-1]
print(list3)          #[5, 4, 3, 2, 1]
# reverse the first 4 numbers
list3 = list2[-2::-1]
print(list3)          #[4, 3, 2, 1]

```

```
# reverse the middle numbers
list3 = list2[-2:-5:-1]
print(list3)          #[4, 3, 2]
```

```
# Create a list pre-initialized with one values of a specified length.
# an list of 10 0's is created
list3 = [0] * 10
print(list3)          # [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

3 ways to copy a list

Make a copy of this list using the copy function:

```
list2 = list1.copy()
```

by slicing:

```
list2 = list1[:]
```

by calling the list constructor:

```
list2 = list(list1)
```

Sorting lists:

```
# sort a list in place
list3.sort()
print(list3)          # [1, 2, 3, 4, 5]
```

```
# sort a list in place descending
list3.sort(reverse=True)
print(list3)          #[5, 4, 3, 2, 1]
```

```
# return a sorted list
list4 = sorted(list3)
print(list4)          # [1, 2, 3, 4, 5]
```

```

# return a sorted list in descending order
list4 = sorted(list3, reverse=True)
print(list4)          #[5, 4, 3, 2, 1]

# add a list to the end of another list
list1 = [1,2,3,4,5]
list2 = [5,6,7,8,9]
list1.extend(list2)
print(list1) # [1, 2, 3, 4, 5, 5, 6, 7, 8, 9]

# join 2 lists together
list1 = [1,2,3,4,5]
list2 = [5,6,7,8,9]
list3 = list1 + list2;
print(list3) # [1, 2, 3, 4, 5, 5, 6, 7, 8, 9]

# remove a value from a list
list2.remove(3);
print(list2)          # [1, 2, 4, 5]

# remove a list item by index
del list2[0]
print(list2)          # [2, 4, 5]

# remove more than 1 item in a list using index slicing
del list2[0:3]
print(list2)          # [5]

# test if a value is in a list returns True or False
x = 2 in list2          # True
print (x)
x = 1 in list2
print (x)              # False

# put a list inside another list
list1.append(list2);
print(list1)          # [5, [2, 4, 5]]

```

```
# convert a string to a list
s = 'happy days are here again'
list1 = s.split(' ')
print(list1)
```

```
['happy', 'days', 'are', 'here', 'again']
```

```
# get a list of letters by calling list constructor with a word
list2 = list("hello")
```

```
['h', 'e', 'l', 'l', 'o']
```

```
# which is quite different then
list2 = ["hello"]
```

```
['hello']
```

```
# convert a list to a string
# *** list must contain all strings ***
list1 = ['happy', 'days', 'are', 'here', 'again']
s = ' '.join(list1)
print(s)
```

```
happy days are here again
```

```
# sum all numbers in a list
list1 = [1,2,3,4,5]
print(sum(list1)) # 15
```



```
# find largest number in a list
```

```
list1 = [1,2,3,4,5]
```

```
print(max(list1)) # 5
```

```
# find smallest number in a list
```

```
list1 = [1,2,3,4,5]
```

```
print(min(list1)) #1
```

```
# find the average of a list
```

```
list1 = [1,2,3,4,5]
```

```
print(sum(list1)/len(list1)) # 3
```

```
# find count of 2's
```

```
list1 = [1,2,2,2,3,5]
```

```
print(list1.count(2)) # 3
```

```
# find most occurrence of number
```

```
list1 = [1,2,2,2,3,5] # 2
```

```
print(max(list1,key=list1.count))
```

```
# find first least occurrence number
```

```
list1 = [1,2,2,2,3,5] #1
```

```
print(min(list1,key=list1.count))
```

```
# convert a list of integers into a list of strings using map
```

```
list1 = [1,2,3,4,5]
```

```
s = list(map(str,list1))
```

```
print(s)
```

```
['1', '2', '3', '4', '5']
```

In this situation we use the python **map** function that takes the function **str** and a **list** of integers as parameters. Using the past str function, the **map** function converts each number to a string. We then convert the result to a list object calling the list constructor.

```
# convert a list of integers to a string of numbers
list1 = [1,2,3,4,5]
s = " ".join(map(str,list1))
print(s)
```

```
1 2 3 4 5
```

In this situation we use the `map` function that takes a the function `str` and a **list** of integers. Using the `str` function and `list1`, the **map** function converts each number to a string, whereas the **join** function joins all the string numbers together from the `map` result.

Using your own function with map

We first make a square function:

```
def sq(x):
    return x*x
```

We use the **square** function with **map** to print out the square of numbers contained in a list. The **map** applies the **sq** function to each item in the list. A list object is then made from the `map` result.

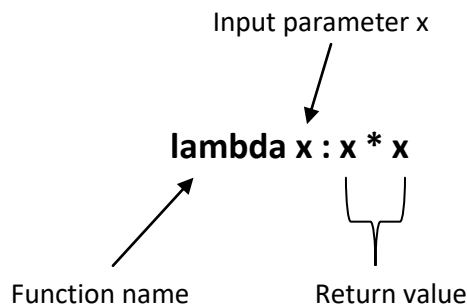
```
list 2 = list(map(sq,list1))
print(list2)
```

```
[1, 4, 9, 16, 25]
```

Alternatively you can use an inline function called **lambda**. A **lambda** function is just a convenient function without a **def** keyword, a name and a parameter list.

normal function	Lambda function
<pre>def sq(x): return x*x</pre>	<pre>lambda x: x* x</pre>

They both do the same thing, the only difference is there is less overhead. The input parameters are on the left of the colon and the return statement is on the right of the colon



We use the **inline lambda** square function with **map** to print out the square of numbers contained in a list. **map** applies the lambda square function to each item in the list.

```
list2 = list(map(lambda x: x* x,list1))
print(list2)
```

```
[1, 4, 9, 16, 25]
```

Executing a function stored in a list

We first make a square function:

```
def sq(x):
    return x*x
```

We store the function in a list with a input argument, so it executes when the list is made

```
list1 = [1,2,sq(3)]
print(list1)
```

```
[1, 2, 9]
```

We store the function name in a list without a input argument

```
list1 = [1,2,sq]
```

we now execute function stored in a list giving the argument 5

```
print(list1[2](5))
```

25

We can also store a lambda inline function in a list (without a input argument)

```
list1 = [1,2,lambda x:x*x]
```

we execute the lambda inline function stored in a list giving the argument 5

```
print(list1[2](5))
```

25

to do:

Make a list of your favourite animals and apply all the list operations on them.

```
animals = ['cat','dog','pig','zebra','elephant']
```

passing a list of values to a function

It is easy to pass a list of values to a function.

```
list1 = [1,2,3,4,5]
```

function to return the sum of numbers in a list

```
def f(list1):  
    total = 0  
    for x in list1:  
        total = total + x  
    return total  
total = f(list1)  
print("the total is: ", total) # 15
```

passing a unknown number of values to a function

We can use the * operator to unpack a many numbers, and treat them like a list of numbers.

function to return the sum of numbers in a list

```
def f(*numbers):  
    total = 0  
    for x in numbers:  
        total = total + x  
    return total  
total = f(1,2,3,4,5)  
print("the total is: ", total) # 15
```

Two Dimensional Lists

Two dimensional lists are analogues to 2 dimensional arrays in other programming languages.

There are 2 ways to make a 2 dimensional list in Python, manually and program ably. We first do manually.

```
list2 = [[1,2,3],[4,5,6],[7,8,9]]  
print(list2
```

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

A 2 dimensional list is arranged as rows and columns. The rows are horizontal and the columns are vertical.

	Column 0	Column 1	Column2
Row 0	1	2	3
Row 1	4	5	6
Row 2	7	8	9

You assign new values to a 2 dimensional array by row and column index, where row indexes start at 0 and column index also start at 0.

```
list_name [row_index][col_index] = value
```

assign value by row and column

```
list3[1][2] = 5
```

You retrieve values also from a 2 dimensional array by row and column index. Where row indexes start at 0 and column index also start at 0.

```
list_name [row_index][col_index] = value
```

retrieve value by row and column

```
x = list3[1][2]
```

```
print(x) # 5
```

To find out how many rows you have you use **len** on the 2 dimensional list. In our 2 dimensional list we have 3 rows.

```
rows = len(list2)
```

```
print(rows) # 3
```

A 2 dimensional list is actually a series of 1 dimensional list's where each row is a 1 dimensional list.

print first row of a 2 dimensional list

```
print(list3[0])
```

```
[1, 2, 3]
```

To find out how many column you have in a particular row you use **len** on that row. In our 2 dimensional list we have 3 columns.

```
cols = len[list2[0]]
print(cols) #3
```

To make a two dimensional list program ably you make a one dimensional list and then assign additional one dimensional lists to it.

```
# make a one-dimensional list of size 3
list3 = [None] * 3;
```

We use **None** as a value because the one dimensional will include another one-dimensional array. (None means nothing or null in other programming languages)

```
# assign one dimensional list's of size 3 to the one-dimensional list
```

```
list3[0] = [0] * 3
list3[1] = [0] * 3
list3[2] = [0] * 3
```

```
print(list3) # [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

```
# assign value by row and column
```

```
list3[1][2] = 5
```

```
# retrieve value by row and column
```

```
x = list3[1][2]
```

```
print(x) # 5
```

```
# print first row of a 2 dimensional list
```

```
print(list3[0]) # [1, 2, 3]
```

Sets

Sets are like lists but only store unique values. The set operations are not as extensive as a list.

```
# To create a empty set
```

```
s1 = set()
```

```
print(s1) # set()
```

```
# To create a list with pre-initialized values
```

```
s2 = {1,2,3,3,4,5}
```

```
print(s2) # {1, 2, 3, 4, 5}
```

get the number of elements in an set

```
x = len(s2)
```

```
print (x)          # 5
```

Add a value to a set

```
s1.add(5)
```

```
print(s1)          # {5}
```

Print a set

```
print(s2)          # {1, 2, 3, 4, 5}
```

try to write a value to a set

```
s[0] = 5
```

You cannot assign a value to a set, python will generate this error:

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

remove a value from a set

```
s2.remove(3)       # {1, 2, 4, 5}
```

```
print(s2)
```

test if a value is in a set

```
x = 2 in s2
```

```
print (x)          # True
```

```
x = 1 in s2        # True
```

```
print (x)
```

update values from another set

```
s1.update(s2)
```

```
print(s1) # {1,2,4,5}
```

intersection of 2 sets

```
s3 = s1.intersection(s2)
```

```
print(s3)          # {5}
```

union of 2 sets

```
s3 = s1.union (s2)
```

```
print(s3)          # {1,2,3,5}
```



```

# put a set in a list
s4 = {1,2,3}
list1 = [s4]
print(list)    # [{1, 2, 3}]

# put a list in a set using the set constructor
s5=set([1,2,3])
print(s5) # {1, 2, 3}

# put a string in a set using the set constructor and get individual letters

S6=set('tomorrow')
print(s6) # {'t', 'w', 'm', 'o', 'r'}

```

Which is quite different from initializing a set with a string

```

S7 = {'tomorrow'}
print(s7) # {'tomorrow'}

```

to do

Make sets s1 and s2 using the set constructor with 2 of your favorite words.

```

s1=set('hello') # {'l', 'e', 'h', 'o'}
s2=set('goodbye') # {'b', 'o', 'g', 'd', 'y', 'e'}

```

Using union and intersection make sets s3 and s4, and print out the results.

```

s3 = s1.union(s2)
print(s3) # {'h', 'b', 'g', 'y', 'l', 'e', 'o', 'd'}
s4 = s1.intersection(s2)
print(s4) # {'e', 'o'}

```

to do:

A Set has many other operations like copy, clear, difference, union, intersection, pop, update etc. You can try these on your own.

Make a couple of sets of your favourite animals.

Example: union would include elements from both sets.

```
s3 = s1.union(s2)
```

Tuples

Tuples store a group of values. They are not as powerful as lists and their main purpose is to return multiple values from functions or represent multiple values when they are needed like in a list. They are read only, meaning you cannot change their values once they are created.

```
# make a empty tuples
```

```
t = ()
```

```
print(t) # ()
```

```
# make a tuple
```

```
t = (1,2,3)
```

```
print(t) # (1,2,3)
```

```
# retrieve a value from a tuple
```

```
x = t[0]
```

```
print(x) # 1
```

```
# print values from a tuple
```

```
print(t[0]) # 1
```

```
print(t[1]) # 2
```

```
print(t[2]) # 3
```

```
# try to write a value to a tuple
```

```
t[0] = 5
```

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

```
#put a tuple in a list
```

```
List1 = [t]
```

```
print(list1) # [(1, 2, 3)]
```

```
# make a list from a tuple  
print(list(t)) # [1,2,3]
```

Example using a tuple to receive values from a function

```
# function that returns value 1,2,3  
def myfunc():  
    return (1,2,3)
```

call function and receive values into tuple variables a,b,c

```
(a,b,c) = myFunc()  
print(a): # 1  
print(b): # 2  
print(c): # 3
```

Dictionaries

Dictionaries store a key and a value. A dictionary can have many keys and corresponding values. Think of a dictionary is like a telephone book with the name as the key and the telephone number as the value.

```
# make empty dictionary
```

```
d = {}  
print (d) # {}
```

```
# add values to a dictionary
```

```
d["name"] = "Tom"  
d["age"] = 24  
d["phone"] = "967-1111"  
print(d)
```

```
{'name': 'Tom', 'age': 24, 'phone': '967-1111'}
```

change a item in a dictionary

```
d["name"] = "Sue"
```

```
print(d)
```

```
{'name': 'Sue', 'age': 24, 'phone': '967-1111'}
```

make dictionary with hardcoded specified keys and values

```
d2 = {"name": "tom", "age": 24, "phone": "967-1111"}
```

```
print (d2)
```

```
{'name': 'Tom', 'age': 24, 'phone': '967-1111'}
```

get values from a dictionary

```
print(d2["name"]) # Tom
```

```
print(d2["age"]) # 24
```

```
print(d2["phone"]) # 967-1111
```

```
Tom  
24  
967-1111
```

#print keys of a dictionary

```
keys = d2.keys()
```

```
print(keys)
```

```
dict_keys(['name', 'age', 'phone'])
```

print values of a dictionary

```
values = d2.values()
```

```
print(values)
```

```
dict_values(['tom', 24, '967-1111'])
```

check if a key is in a dictionary

```
print("Tom" in d2)
```

```
True
```

remove an item in a dictionary by key

```
del d2["name"]
```

```
print(d2)
```

```
{'age': 24, 'phone': '967-1111'}
```

```
# update a dictionary from another dictionary
d2.update({"email": "tom@mail.com"})
```

```
{'name': 'tom', 'age': 24, 'phone': '967-1111', 'email': 'tom@mail.com'}
```

```
# clear all items in a dictionary
d2.clear()
print(d2)
```

```
{}
```

Type all the above examples in your file lesson4.py and, make sure you get the same results. We will be using the lists, tuples and dictionaries in future lessons.

HOMEWORK 5

Question 1

Make an list of your favorite animals like: elephant, cat and dog etc.
Print out the list of animals.

Ask the user of your program to type in one of the animals from your list, that they don't like.

Then ask them to type in an animal (not in the list) they do like.

Replace the animal in the list they don't like with an animal they do like. Then reprint the animal list with the heading "Animals I like".

Make a second list and put in the animal they don't like and the new animal they do like. Print out the second list with the heading "Animals I like and don't like". You need to use the list **remove** and **append** methods.
Put your code in a file called homework5.py

Question 2

From the animal list from question 1 put the animal lists in sets.

```
set1 = set(animals)
set2 = set(animals2).
```

Make another set that combines all the animals together that are liked.
Print out this set. (hint use union set1 and set2)

Make another set that combines all the animals that are not liked.
Print out this set. (hint use intersection set1 and set2)

Put your code in a file called homework5.py

Question 3

Make a dictionary d1 of your favorite animal kinds like (cat, tiger, zebra). Give each animal a name (Fluffy, Sally, Rudolf) . Use the animal name as the key and the animal kind as the value.

Example: key: fluffy value: cat

Then make another dictionary d2, use the same animal kinds (cat, tiger, zebra) as the key and the animal sound (meow, roar, heehee) as the value.

Example: key: cat value: meow

Print out the keys (animal name's) of the first dictionary.

Ask the user to type in one of the animal names.

Get the animal kind from the first dictionary using the animal name.

Print to the screen the name of the animal and what kind of animal it is like:

“Fluffy is a cat”

As the user what sound the animal makes.

What sound does a cat make?

From the second dictionary use the animal kind to get the sound that the animal makes.

Then tell the user what sound the animal makes like “Cat’s meow”

Put your code in a file called homework5.py

Question 4

Change Question 3 so that when the user types in the name of one of the animals, the program asks what kind of animal it is.

For example if the user selects “fluffy”

Then the program asks,

What kind of animal is “fluffy” ?

If they guess correctly then ask the user what sound does a cat make?

If they guess correctly congratulate them.

If they do not guess correctly then just tell them “cat’s meow”

If they do not guess the kind of animal correctly then just tell them

“fluffy is a cat”;

Put your code in a file called homework5.py

LESSON 6 PROGRAMMING STATEMENTS

Programming statements allow you to write complete Python Program scripts. We have already looked at simple input, print and assignment statements. We now need to control the flow of our program. Branch control statements allow certain program statements to execute and others not. Loop control statements allow program statements to repeat themselves.

Start a new python program lesson6.py to test all the control statements

Branch Control Statements

Branch control statements control program flow, analogous from choosing which road to follow when there are many different choices to choose from.

if statement

The **if** branch **control** statements use conditional operators from the previous lessons to direct program flow.

If condition :

Statement(s)

When the condition is evaluated to be true the statements belonging to the if statement will execute.

```
# if statement
```

```
x = 5
```

```
if x == 5:
```

```
    print("x is 5")
```

```
x is 5
```


if with a else statement

We now add an else statement. An if-else control construct is a two-way branch operation.

If condition :

statements

else:

statements

if – else statement

x = 2

if x == 5:

print("x is 5")

else:

print ("x is not 5")

x is not 5

elif statement

We can also have extra else if statements to make a multi-branch. Python contracts else if to **elif**

multi if else

x = 10

if x == 5:

print("x is 5")

elif x < 5:

print("x less than 5")

elif x > 5:print("x greater than 5")

print("I like Python Programming")

x greater than 5

Our multi branch if-else can also end with an else statement.

```
# multi if-else else
x = 5
if x < 5:
    print("x less than 5")
elif x > 5:
    print("x greater than 5")
else:
    print("x is 5")
```

x is 5

nested if statements

if statements can be nested to make complicated conditions simpler

```
# nested if statement
x = 6
if x >= 0:
    if x > 5:
        print("x greater than 5")
    else:
        print("x less than equal 5")
```

x greater than 5

while loop

Our next control statement is the while loop

```
while condition:  
    statement(s)
```

The while loop allows you to repeat programming statements repeatedly until some condition is satisfied

```
# while loop  
x = 0  
while x <5:  
    print(x)  
    x+=1
```

```
0  
1  
2  
3  
4
```

to do:

change **while** loop to print out 1 to 5

change **while** loop to print out 5 to 1

for loop

The other loop is the for loop. It is much more powerful than the while loop but can be difficult to use.

All loops must have count mechanism. The for loop uses the range function that supplies a range of numbers like 0,1,2,3,4 having a start value, end value and step value. Example: **range(0,5)** starts at 0 ends at 4 and increments by 1.

```
for counter in range(start_value, end_value-1, increment):  
    statement(s)
```

The default value for increment is 1. The default start value is 1.

Using a for loop to loop 1 to 5 using the range function. The i variable cannot be changed and belongs to the for loop. The loop stops at 5 not 6 because the stop is end_value – 1.

for loop using range

for i in range(0,5):

print (i)

```
0
1
2
3
4
```

same as

for i in range(5):

print (i)

```
0
1
2
3
4
```

to do:

change **for loop** to print out 1 to 5

Here is a for loop that counts backwards using a negative increment

for loop using range counting backward

for i in range(4,-1,-1):

print (i)

```
4
3
2
1
0
```

change **for loop** to print out 5 to 1

Nested for loops

Nested for loops are used to print out 2 dimensional objects by row and column

```
# nested for loop
for r in range(1,6):
    print(r, ":", end="")
    for c in range(1,6):
        print(c,end=" ")
    print("")
```

Note we use the **end** directive so we do not start a new line every time we print out a column value

```
1 : 1 2 3 4 5
2 : 1 2 3 4 5
3 : 1 2 3 4 5
4 : 1 2 3 4 5
5 : 1 2 3 4 5
```

Loops can also be used to print out characters in a string variable

```
# print out characters in a string using range
s = "Hello"
for i in range(len(s)):
    print(s[i])
```

```
H
e
l
l
o
```

print out characters in a string using in operator

```
s = "Hello"
```

```
for c in s:
```

```
    print(c)
```

```
H
e
l
l
o
```

todo:

print our string backwards

We also can print out the values in a list

For loops can also print out values from lists

print out values in a list using range

```
list1 = [1,2,3,4,5]
```

```
for i in range(len(list1)):
```

```
    print (list[i])
```

```
1
2
3
4
5
```

We also can print out the values in a list

For loops can also print out values from lists

print out values in a list using in operator

```
list1 = [1,2,3,4,5]
```

```
for x in list1:
```

```
    print (x)
```

```
1
2
3
4
5
```

Here we print out values from a two-dimensional array

```
# print out two-dimensional list
```

```
list2 = [[1,2,3],[4,5,6],[7,8,9]]
```

```
for r in list2:
```

```
    for c in r:
```

```
        print (c,end=" ")
```

```
print("")
```

```
1 2 3
```

```
4 5 6
```

```
7 8 9
```

For loops can also print out values from a set

```
# print out values in a set
```

```
set1 = {1,2,3,3,5}
```

```
for x in set1:
```

```
    print (x)
```

```
print("")
```

```
1
```

```
2
```

```
3
```

```
5
```

Put all letters from a string and put into a set

```
s = "tomorrow"
```

```
print(s)
```

```
# make empty set
```

```
set1 = set()
```

```
# put characters in set
```

```
for c in s
```

```
    set1.add(c)
```

```
# print out characters in set
for x in set1:
```

```
    print (x,end= ' ')
print("")
```

```
tomorrow
mrtwo
```

We print out all unique letters of the word stored in the set.

For loops can also be used print out dictionaries. We would want to print out the dictionary in order by key or in order by values.

```
# make dictionary
```

```
d = {"name": "Bob", "age": 24, "studentid": "S1234"}
```

```
# print out dictionary
```

```
for key,value in d.items():
```

```
    print (key, " : ",value)
```

```
name : Bob
age : 24
studentid : S1234
```

Note: **items()** actually returns a list of (key, value) pair tuples.

```
dict_items([('name', 'Bob'), ('age', 24), ('studentid', 'S1234')])
```

This allows the **key** and **value** in the **for** loop to iterate as key and value.

```
# print dictionary by key
```

```
for key in d.keys() :
```

```
    print (key)
```

```
name
age
studentid
```



```
# print dictionary by value
for value in d.values() :
    print (value)
```

```
Bob
24
S1234
```

```
# print dictionary sorted by key
for key in sorted(d.keys()):
    print (key, ":",d[key])
```

```
Age : 24
Idnum : S1234
Name : Bob
```

```
# print dictionary sorted by value
for value in sorted( d.values(), key=str ):
    for key in d.keys():
        if d[ key ] == value:
            print (key, ":",value)
            break
```

```
age: 24
studentid: Bob
name: s1234
```

We used the sorted function to return a list of sorted dictionary values.

Note: we use `key=`**str** option on the dictionary values to convert all elements to string data type while sorting. Our values may have different data types string and int. We need to convert all data values to a string data type for sorting. We pass the `str` function to the sorted function as the sort key. (The sort key is different from dictionary key)

The above loop to sort a dictionary by value is a little inefficient. A more efficient way is to pass a comparison function to the **sorted** function so it knows which elements to sort, sort on key or sort on value

sort dictionary by value using a comparison function

The **fcmp** function receives a (key, value) pair tuple from the dictionary and either returns the **key** x[0] or returns the **value** x[1] enclosed in a optional **str** function.

```
def fcmp(x):  
    return str(x[1])
```

The **sorted** function receive list of dictionary items as tuples and the **fcmp** sort comparison function as a sort key function. We set the key parameter of the **sorted** function to our comparison function **fcmp: key=fcmp** (Note: The key parameter is different from dictionary key)

```
sort_by_value = sorted(d.items(), key=fcmp)
```

The **fcmp** function receives the (key,value) tuples as the **x** parameter of the fcmp function from the dictionary one by one.

fcmp(x)

where :

```
x = ('name', 'Bob')  
x = ('age', 24)  
x = ('studentid', 'S1234')
```

To sort by key then fcmp uses **x[0]** to return the **key** for comparison like 'name';

To sort by value the **fcmp** function uses **x[1]** to return the **value** for comparison like 'Bob'

It all depends what the fcmp function returns to what get sorted. Sort by dictionary key or sort by dictionary value.

The sorted function returns a list of sorted dictionary tuple items, that we can print out.

```
print(sort_by_value)
```

```
[('age', 24), ('name', 'Bob'),('studentid', 'S1234')]
```

We are actually sorting the array of dictionary items of tuples

```
print(d.items)
```

```
dict_items([('name', 'Bob'), ('age', 24), ('studentid', 'S1234')])
```

sort using the sorted function

The comparison **fcmp** function instructs the **sorted** function to sort on the values as a string

```
str(x[1])
```

if you want to sort on the key you would use:

```
str(x[0])
```

Again we use the **str** function to convert all items values to a string so we can sort the same data types.

For convenience we can also use an inline function instead of a separate function. Inline functions are known as an anonymous function called **lambda**. An anonymous function is a function with no name and has arguments and an expression to evaluate using the arguments.

lambda arguments : expression

The expression is executed and the result is returned:

For example:

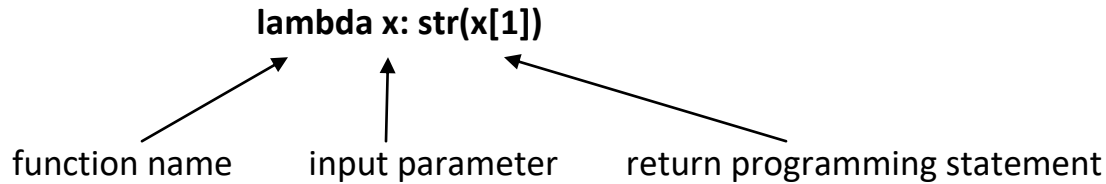
```
x = lambda a, b : a * b  
y = x(3, 4) # 3 * 12  
print(y) # 12
```

```
lambda (a,b):  
    return a * b
```

lambda arguments : expression

*a,b a * b*

Our lambda compare function would be:



Here is the code to sort a dictionary using the sorted function and a lambda anonymous comparison function

```
sort_by_value = sorted(d.items(), key=lambda x: str(x[1]))
```

it returns a list of sorted name, value pair tuples

```
[('age', 24), ('name', 'Bob'),('studentid', 'S1234')]
```

we are actually sorting the array of dictionary items tuples

```
dict_items([('name', 'Bob'), ('age', 24), ('studentid', 'S1234')])
```

using the sorted function, the lambda function receives one of the tuples `x` and return the string value `x[1]`

note: if we did not have mixed data types then we could use `x[1]` not `str(x[1])`

```
sort_by_value = sorted(d.items(), key=lambda x: x[1])
```

To do:

Make sure you try all the loop statements and are working before proceeding. .
Make a dictionary of all your favorite animals using the animal type as the key and the animal name as the value. Print out the dictionary sorted by name value. Next make a dictionary of all your favorite animals using the animal name as the key and the animal type as the value. Print out the dictionary sorted by animal type value.

LESSON 6 HOMEWORK

Put all your homework code in a homework6.py file

Exam Grader

Ask someone to enter an exam mark between 0 and 100. If they enter 90 or above printout an "A", 80 or above print out a "B", 70 or above print out a "C", 60 or above print out a "D" and "F" if below 60. Hint: use if else statements.

You can visualize a grade chart like this:

Mark Range	Exam Grade
90 to 100	A
80 to 89	B
70 to 79	C
60 to 69	D
0 to 59	F

Bonus: validate the entered mark between 0 and 100, if they are out of bounds ask them to re-enter a correct number between 0 and 100.

Make a loop so they can enter marks continuously, use -1 to exit

Print out a comment from the following chart for each exam grade they get.

Mark Range	Exam Grade	Comment
90 to 100	A	"Excellent",
80 to 89	B	"Good",
70 to 79	C	"Okay",
60 to 69	D	"Print Study Harder"
0 to 59	F	"Talk to your teacher"

Mini Calculator

Make a mini calculator that takes two numbers and a operation like -, +, * and /.
Prompt to enter two number's and a operation like this:

Enter first number: 3
Enter second number: 4
Enter (+, -, *, /) operation: +

Then print out the answer like this:

$3 + 4 = 7$

Hint: if elif else statements

The else statement should be used to indicate a invalid operation

Use a while or do while loop so that they can repeatedly enter many calculations.

Terminate the program when they enter a letter like 'X' for the first number.

Hint: use upper or lower to look for the 'X'

Triangle Generator:

Use nested for loops to print out a triangle using '*' like this:

```
      *
     * *
    * * *
   * * * *
  * * * * *
 * * * * * *
```

Ask the user how many rows they want, validate your input, you cannot have negative rows.

Hint: use 2 nested for loops, start with a square of stars
To print out a character like '*' or ' ' use

```
print('*', end='')
```

to print a new line use:

```
print("")
```

Enhanced Triangle Generator:

Use nested for loops to print out a triangle using '*' like this:

```
      *
     * * *
    * * * * *
   * * * * * * *
  * * * * * * * *
 * * * * * * * * *
```

Ask the user how many rows they want.

Hint: use 2 nested for loops, start with a square of stars

Reverse a String

Reverse a String using a while loop or a for loop. You will have to put the reversed characters in a second string or in the same string using slicing. You cannot in Python directly change the characters in the original String.

Test if a number is prime

Make a function called **isPrime(x)** that tests if a number is prime. A prime number is a number that can be only divided by 1 and itself. In a loop divide the number between 2 to number-1 (or 2 to square root of number+1. For square root use:

```
import math
x = int(math.sqrt(n))
```

If the number can be divided by any of the divisors then the number is not prime, else it is prime. Use the % (mod) operator to test if a number is evenly divided by another number. Example: $x \% i == 0$. Don't forget to start from number 2. Print out the first 100 prime numbers.

The first 10 prime numbers: 2, 3, 5, 7, 11, 13, 17, 19, 23, and 29

Print out all factors of a number

Make a function call **factors(x)** that will print out all the factors of a number. The factors of a number is all the divisors divided by the number evenly. Ask the user of your program to enter a number then print out all the factors.

Example:

Enter a number: 50

The Factors of 50 are:

1
2
5
10
25
50

Print out all prime factors of a number

Make a function call **prime_factors(x)** that will print out all the prime factors of a number. The prime factors of a number is all the prime number divisors divided by the number evenly.

Example: $12 = 2 \times 2 \times 3$

Following are the steps to find all prime factors.

- 0) Enter a number n
- 1) While n is divisible by 2 (use & 2), print 2 and then integer divide n by 2
 $n = \text{int}(n//2)$
- 2) In a **for** loop from $i = 3$ to square root of $n + 1$ and increment by 2 use $\text{int}(\text{math.sqrt}(n+1))$
in a **while** loop while n is divisible by i (use mod %)
print i
integer divide n by integer i $\text{int}(n//i)$
- 3) print n if it is greater than 2.

For square root use:

```
import math  
x = int(math.sqrt(n+1))
```

Example using:

```
Enter a number: 12  
The prime factors of 12 are:  
2 2 3
```

Make a Guessing game

Ask the user of your game to guess a number between 1 and 100. If they guess too high tell them “Too High”. If they guess too low tell them they guess “Too Low”. If they guess correct tell them “Congratulations you are Correct!”. Keep track in a list how many tries the user took. At the end of the game, print out the number of games played and the average tries of all games if they played more than 1 game. Ask the user if they want to play another game.

You will first need to generate a random number to guess.

You can use this code to generate a random number:

```
x = random.randint(1, MAX_NUMBER)
```

Where **MAX_NUMBER** is a constant placed at the top of your program.

```
MAX_NUMBER = 100
```

You will need to include the following at the top of your program, for python to recognize the **randint** function.

```
import random
```

Also make another constant **MAX_TRIES** for the number of tries allowed.

```
MAX_TRIES = 10
```

You should have functions to print a welcome message explaining how to play the game, generate a random number, get a guess from the keyboard, check if a guess is correct and print out the game scores. The main function should just call your functions in a loop. Call your python file `homework6.py` or `GuessingGame.py`

Object Oriented Guessing Game

Make a **GuessGame** class that will keep track of the guess number and tries per game and class variables total tries and number of games. You should have methods to generate a random number, check if a guess is correct, too low or too high, increment tries, return the score per game and class methods to calculate and return average score of all games and the number of games played.

The Game **__init__** method will generate and store the **random number** to guess, and **tries** per game.

Class variables are placed right after the class definition. Class variables are shared between all instances, so that they will have the same value for all objects. Class variables are declared without the self keyword.

Class Game:

```
TotalTries = 0
```

```
NumGames = 0
```

You will also need to make a class function to return the average score and numGames. Class functions do not have the self keyword

```
# calculate and return average score  
def getAverageTries():  
  
    return TotalTries / NumGames  
  
# return number of games  
def getNumGames():  
  
    return TotalTries// NumGames
```

You call the class functions using the class name not the object variable.

```
print("total games",GuessGame.getNumGames())  
print("average score",GuessGame.getAverageScore(),tries)
```

The main function should just handle inputs from the keyboard and printing output to the console. The **GuessGame** class should not handle any input and output, and is used, mainly to store the random number and tries per game. The main function would instantiate a new GuessGame object per game. After each game is played ask the user if they want to play another game.

After all games have been played print the average tries per game. Call your python file homework6.py or GuessingGame2.py

Sentence Generator

A Sentence is composed of the following:

<article><adjective><noun><adverb><verb><article><adjective><noun>

Make a list of articles like: 'a', 'an' and 'the'

```
articles = ['a', 'an', 'the']
```

Then make a list of adjectives like: 'fat', 'big', 'small'

Then make a list of nouns like: 'cat', 'rat', 'house'

Then make a list of adverbs like: 'slowly', 'gently', 'quickly'

Then make a list of verbs like 'ate', 'sat on', 'pushed'

EACH LIST SHOULD HAVE THE SAME AMOUNT OF ENTRIES LIKE 3 ENTRIES EACH

Make a dictionary to hold all the lists:

```
words = {'articles':articles, 'adjectives':adjectives, 'nouns':nouns,  
'adverbs':adverbs, 'verbs':verbs}
```

Next make a list of dictionary keys to make a sentence:

```
keys = ['articles', 'adjectives', 'nouns', 'adverbs', 'verbs', 'articles', 'adjectives', 'nouns']
```

Finally make a sentence using the dictionary entries, using the key list and by selecting random words from the dictionary list values.

```
import random
```

```
sentence = ""
```

```
for key in keys:
```

```
    sentence += words[key][random.randint(0,2)] + " "
```

Then print out the sentence. You should get something like this:

```
print(sentence)
```

```
The big cat slowly ate the small rat
```

Which has picked random words from the dictionary sentence structure:

<article><adjective><noun><adverb><verb><article><adjective><noun>

You can put your code in your homework6.py file or make a sentence.py file

LESSON 7 File I/O

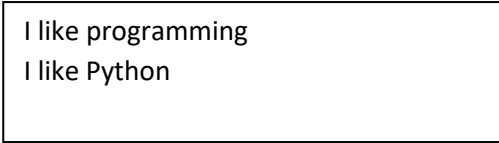
File Access

Files store data that can be read and written to. Make a Leson6.py file to store all the following python statements.

Writing lines to a file

We first write lines to a file so that we can read it back. We use the open method to open the file using the "w" file mode specifier and then use the write method to print a line to the file. We then use the close method to close the file. If you do not close the file you will lose all the data written to the file.

```
# write lines to a file
f = open("test.txt", "w")
f.write("I like programming")
f.write("\n")
f.write("I like Python")
f.write("\n")
f.close()
```



```
I like programming
I like Python
```

“\n” is the end of line terminator, alternatively you can put the “\n” at the end of each line you write like this:

```
f.write("I like programming\n")
```

reading lines from a file

We then read the file back and print it to the screen. We use the open method to open the file with the "r" file mode specifier and then use the **readlines** method to read the lines from the file. We use the **close** method to close the file. If you do not close the file then the file may not be able to be used by other programs.

```
# Open the file for read
f = open('test.txt', "r")

# read all lines in the file to a list
lines = f.readlines()
print(lines)

# close the file
f.close()
```

```
['I like programming\n', 'I like Python\n']
```

Note: **readlines** actually returns a list of lines like:

```
['I like Programming\n','I like Python\n']
```

We can use a for loop to read each line from a file traversing through the **readlines** function

```
# Open the file for read
f = open('test.txt')

# read line one at a time
# till the file is empty
for line in f.readlines():

    print (line)

f.close() # close file
```

```
I like programming
i like Python
```

When you print out each line in the list you can use the function **strip** to remove the “\n” end of line character, so you do not get empty lines.

```
print (line.strip())
```

We can also read the file line by line using the **readline** function

```
# Open the file for read
f = open('test.txt')

# Read the first line
line = f.readline()

# read line one at a time
# till the file is empty
while line:

    print (line)
    line = f.readline()

f.close() # close file
```

```
I like programming
I like Python
```

writing lines to end of a file (appending lines to a file)

You can also write lines to end of an existing file use the **"a"** file mode specifier

```
# write lines to end of a file
f = open("test.txt", "a")

f.write("I like Python")

f.write("\n")

f.close()
```

```
I like Python
```

read the file again

```
# Open the file for read
f = open('test.txt')

# read line one at a time
# till the file is empty
for line in f.readlines():

    print (line)

f.close() # close file
```

```
I like programming
I like Python
```


Write lines to a csv file.

A csv file is known as comma separated values and are used to store data by rows and commas.

```
# write lines to end of a file  
f = open("test.csv", "w")  
f.write("one,two,three,four")  
f.write("\n")  
f.close()
```

Read lines from a csv file.

A csv file is known as comma separated values and are used to store data by rows and commas. We read a line from the file using the readline function then use the strip method to remove the end of line character and then use the split function to separate the words between the commas. The words are placed in a list called tokens.

```
# read csv file  
  
# Open the file for read  
f = open('test.csv')  
  
# Read the first line  
line = f.readline()  
  
# read line one at a time  
# till the file is empty  
while line:  
  
    # strip removes '\n'  
    # split separates line into tokens  
    tokens = line.strip().split(",")  
  
    # print list of tokens  
    print (tokens)
```

```
['one,two,three,four']  
  
one  
two  
three  
four'
```

```
# printing out individual token
for t in tokens:

    print(t)

line = f.readline()

f.close()
```

Catch file error

You need to catch a error when a file cannot be opened. If you do not then program will stop working. The **try – except** block will catch and report the file error.

```
try:
    f = open('test.txt', 'r')
    for line in f.readlines():
        print(line)
    f.close()

except IOError:
    print ('cannot open file test.txt')
```

Writing Object to Files

You can write objects to files using **pickle**, this is called serialization.

We first make a book class that we can be later written out to a file.

class Book:

```
def __init__(self, title, author):

    self.title = title
    self.author = author

def __str__(self):

    return "Book: " + self.title + " written by: " + self.author
```

We then make book object from the book class definition:

```
b = Book("Wizard of Oz", "L. Frank Baum")
```

Now we write book object to a binary file called book.p using **pickle.dump**. You must import pickle before you can use it.

```
import pickle  
f = open( "book.p", "wb" )  
pickle.dump( b, f )  
f.close()
```

We then read the book back from file using **pickle.load**.

```
f = open( "book.p", "rb" )  
book = pickle.load( f )  
f.close()
```

We then print out the book object.

```
print(book)
```

```
Book: Wizard of Oz written by: L. Frank Baum
```

LESSON 7 HOMEWORK

Question 1

Open a file for write and write a small story to it of between 5 to 10 lines. Call your file "story.txt"

Using **try** and **except** open the text file and count the number of letters, word, sentences and lines it. Words are separated by spaces and new lines. Sentences are separated by periods “.” or other punctuation like “?”.

Lines are separated by ‘\n’. Words may contain numbers and punctuation like apple80 and don’t. You can separate a line read from a file into words using the **split** function. You can remove punctuation on a word by using the replace function.

```
word = word.replace(",","")
```

Keep track of each word count in a dictionary.

Write a report to a file called report.txt, the number of letters, words, sentences and lines. Use the ‘f’ formatter to write each report line to the file.

```
s = f'Number of words: {numwords}\n'
```

You just put the variable that store’s the value in { } brackets. Print out the words and word count in descending order.

You can use:

```
sorted_words = sorted(wordList.items(), key=lambda x: x[1], reverse=True)
```

Open the report file and display the report file lines to the screen. Call your python file homework7.py.

Question 2

Write a program that writes out another python program to a file. Then open up the file you wrote that contains the python program and execute it.

Algorithm:

1. Open up a file for write with a py extension like “test.py”
2. Write lines to a file with a input statement or print statements like:
3. f.write(“print(‘I like python’)\n”)

you need to alternate between single ‘ quotes and double “ quotes

4. close the file
5. open the test.py file in your python IDE and run the program

It should print out

```
I like programming
```

Call your python program 'homework7q2.py'

Bonus:

Add more print , input, etc python statements, you could even write out this question code so you have a program the writes out itself!

LESSON 8

LIST COMPREHENSION, ITERATORS, GENERATORS AND HIGHER ORDER FUNCTIONS

List Comprehension is an easy way to append items to a list. The syntax is a little complicated to understand, but if you accept that it works, then this is the best approach to take. It is best to do things first, and understand later.

The syntax is very intimidating, but **List Comprehension** is very powerful

```
new_list = [ expression for loop iteration ]
```

List Comprehension always returns a new list, the *expression* is the **value** that is appended to the *new list* and the *loop iteration* is used to specify how many iterations that are going to take place.

Here is a List Compression example that builds a list with the values 1 to 10.

```
new_list = [i for i in range(10)]
```

```
print (new_list)
```

would print out

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

In our above example the expression is **i** and for loop iteration is:

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

which is the same **i** in the expression

```
new_list = [ expression for loop iteration ]  
new_list = [ i for i in range(10) ]
```

This list comprehension statement is equivalent to:

```
new_list = []
for i in range(10)
    new_list.append(i)
```

We can expand our expression to do some calculation like square root

```
new_list = [i*i for i in range(10)]
print (new_list)
```

would print out

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

We can add a **filter condition** to our List Comprehension to print out even number squares. Our **filter condition** is made using an **if** statement.

```
new_list = [ expression    for loop iteration    filter condition]
```

The condition is applied to the *x* in the iteration for loop, not to the output value appended to the output list.

```
# print out even square numbers
squares = [x*x for x in range(10) if x % 2 == 0]
print (squares)
```

would print out:

```
[0, 4, 16, 36, 64]
```

```
new_list = [ expression    for loop iteration    filter condition]
squares = [    x*x          for x in range(10)    if x % 2 == 0]
```

To do:

Write a List Comprehension to print out odd number squares

Doing list compression on another list

Here is a List Comprehension example that does operation on another list

The syntax is

```
new_list = [ expression for item in old_list ]  
  
old_list = [1, 2, 3, 4, 5]  
new_list = [x*2 for x in old_list]  
print(new_list)
```

would print out

[2, 4, 6, 8, 10]

This example basically iterates through the old_list item by item and each item is multiplied by 2 and then appended to the new list.

The List Comprehension statement is equivalent to:

```
old_list = [1, 2, 3, 4, 5]  
new_list = []  
for x in old_list  
    new_list.append(x * 2)  
print(new_list)
```

would print out:

[2, 4, 6, 8, 10]

Here is a list example to add a condition to filter to a input list. We will print out the even number times 2 of our input list applying a condition filter to it. The condition is applied to the values of the old list, not to the output of the new list

The syntax is

```
new_list = [ expression for item in list if conditional ]
```

```
old_list = [1, 2, 3, 4, 5]  
new_list = [x*2 for x in old_list if x % 2 == 0]  
print(new_list)
```

would print out:

```
[4, 8]
```

<pre><i>new_list = [expression for item in list if conditional] new_list = [x*2 for x in old_list if x % 2 == 0]</i></pre>
--

To do:

Write a List Comprehension to print out the odd numbers times 2.

List Comprehension tuples

The output list may also contain tuples

```
# print list of tuples  
old_list = [1, 2, 3, 4, 5]  
new_list = [(x,x*2) for x in old_list]  
print(new_list) # [(1, 2), (2, 4), (3, 6), (4, 8), (5, 10)]
```

```
# print even tuples  
old_list = [1, 2, 3, 4, 5]  
new_list = [(x,x*2) for x in old_list if x % 2 == 0]  
print(new_list) # [(2, 4), (4, 8)]
```

to do

print a list of odd tuples squared

```
[(1, 1), (3, 9), (5, 25)]
```

List Comprehension examples

List Comprehension has many uses like: initializing lists with values, converting values etc. Here are examples to create a 1-dimensional array with List Comprehension

```
oneD_array = [0 for i in range(3)]
print(oneD_array)
[0, 0, 0]
```

Here are examples to create 2-dimensional array with List Comprehension

```
num_rows = 2
num_columns=3
twoD_array = [[0 for i in range(num_columns)] for j in range(num_rows)]
print(twoD_array)
[[0, 0, 0], [0, 0, 0]]
```

Here is another example to sort a list of mixed data types using List Comprehension

```
mixed_list = ['a',3,'b',5]
sorted_list = sorted([str(x) for x in mixed_list])
print(sorted_list) # ['3', '5', 'a', 'b']
```

It basically converts each element in the mixed list to a string so that the sorted function can sort strings, The sorted function cannot sort mixed data types at the same time string and ints.

Using List Comprehension to remove '\n' from a list of strings

```
list1 = ["cat\n","dog\n","lion\n"]
print(list1)
```

```
['cat\n', 'dog\n', 'lion\n']
```

```
list2 = [x.replace("\n", "") for x in list1]
```

```
print(list2)
```

```
['cat', 'dog', 'lion']
```

In this example the '\n' has been replaced by a "" which is an empty string.

HOMework

Question 1

Use List Comprehension to convert a list of five string numbers to integers and return a sorted list of integer numbers

Example

Before:

```
['3', '5', '7', '2', '6']
```

After:

```
[2, 3, 5, 6, 7]
```

Put your homework in a Python file called homework9.py

Iterators

Iterators allow you to traverse through a collection value by value. A collection may be a list, set or etc of values. We have already traversed through a list using a for loop. A list is **iterable** meaning it automatically returns an iterator object for traversals. In the following code the for loop uses the iterator of the fruits list to traverse through the fruit list element by element,

```
fruits = ['apple','orange','banana','pear']
```

```
for fruit in fruits:
```

```
    print(fruit)
```

```
apple
```

```
orange
```

```
banana
```

```
pear
```

An **iterator** has methods `__iter__()` and `__next__()`.

The `__iter__()` method creates an iterator and the `__next__()` method retrieves the next value, as the iterator traverses through the values.

In the following example we use a iterator to traverse the fruit list fruit by fruit.

```
itr = iter(fruits)
```

```
for i in range(len(fruits)):
```

```
    print(next(itr))
```

```
apple
```

```
orange
```

```
banana
```

```
pear
```

The following **for loop** automatically does the same thing as above.

```
for fruit in fruits:
```

```
    print(fruit)
```

The for loop actually creates an iterator object and executes the `next()` method for each loop.

making your own iterator

We will make an iterator that returns a sequence of square numbers.

```
class SquareIterator:
```

```
    def __iter__(self):
```

```
        self.x = 1
```

```
        return self
```

```

def __next__(self):
    if self.x <= 10:
        x = self.x
        self.x += 1
        return x*x
    else:
        raise StopIteration

```

The `__iter__` method initializes the **SquareIterator** with x starting at 1. The `next()` method returns the square number then increments x for the next square number. When the square iterator finishes its sequence the **StopIteration** exception is raised. The for loop will catch the StopIteration exception and stop the for loop. We run the Squares Iterator like this:

```

# make and run iterator
itr = iter(SquaresIterator())
for x in itr:
    print(x)

```

```

1
2
4
9
16
25
36
49
64
81
100

```

Alternatively you can just call the next in a loop and exit the loop when the **StopIteration** exception is thrown like this:

```

g = squareGenerator()
while(True):
    try:
        print(next(g))
    except StopIteration:
        break

```

```

1
2
4
9
16
25
36
49
64
81
100

```

This iterator only works on python version 3.x

Here is the version for python version 2.7. The `__next__()` method has been replaced by `next()` (without the underscores `__`)

```
class SquareIterator:
```

```
    def __iter__(self):
```

```
        self.x = 1
```

```
        return self
```

```
    def next(self):
```

```
        if self.x <= 10:
```

```
            x = self.x
```

```
            self.x += 1
```

```
            return x*x
```

```
        else:
```

```
            raise StopIteration
```

```
# make and run iterator
```

```
itr = iter(SquareIterator())
```

```
for x in itr:
```

```
    print(x)
```

```
1
2
4
9
16
25
36
49
64
81
100
```

Alternatively you can run the iterator in while loop calling the `next` method and `break` the loop when the `StopIteration` exception is throw

```
itr = SquareIterator()
```

```
while(True):
```

```
    try:
```

```
        print(itr.next())
```

```
    except StopIteration:
```

```
        break
```

```
1
2
4
9
16
25
36
49
64
81
100
```

to do

Change the **SquareIterator** to a **SquareRoot** Iterator.

LESSON 8 HOMEWORK Part1

Question 2

Make a iterator that receives a list of numbers and return the numbers in list in reverse. You will need to make a **__init__()** function that receives a list

Call your iterator **ReverseIterator**.

Put your homework in a python file called homework9.py

Generators

A Generator uses the **yield** statement that suspends a function's execution and then send's back a value to the caller through a **iterator**. Each value is retrieved by calling the **next ()** method from the iterator. The function then resumes execution and will run until another yield statement is executed.. This allows the function to produce a series of values over time. The values are retrieved from the function one by one by calling the **next** method from the returned iterator. For every iterator **next** method call a value is returned and the function resumes execution until the next yield is encountered and suspends again. Any function with a **yield** statement becomes a generator.

Simple generator

This simple generator just returns the value **x** when the **next** method of the iterator is called.

```
def simpleGenerator():
```

```
    x = 5
```

```
    yield x
```

The **simpleGenerator** returns an iterator of values

```
g = simpleGenerator()
```

We iterate through the return iterator by calling the **next** method and print out the values. In this case it is the value 5

```
x = next(g)  
print (x)
```

```
5
```

Alternatively you can use a for loop instead. You need to make another generator because all the values of the previous generator have all been read.

```
g = simpleGenerator()
```

```
for x in g:  
    print (x)
```

```
5
```

A generator may have more than 1 yield statement

```
def simpleGenerator2():
```

```
    x = 5  
    yield x  
    x += 5  
    yield x
```

The **simpleGenerator2** returns a value for each iterator next method called.

We run the generator like this:

```
g = simpleGenerator2()
```


You can get the values from the generator for each **next** method called

```
x = next(g)
print (x)
x = next(g)
print (x)
```

```
5
10
```

We iterate through the return iterator in a for loop and print out the values.

```
for x in g:
    print (x)
```

```
5
10
```

Square Generator

The square generator generate a sequence of squared values. The function suspends operation when the **yield** statement is encountered, and the function resumes execution when the value is retrieved by calling the **next** method.

```
def squareGenerator(n):
    for x in range(n):
        yield x*x
```

Here we call the squareGenerator the returns an iterator object.

```
g = squareGenerator(10)
```

print out values using the next method call

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

```
1
2
4
9
16
25
36
49
64
81
100
```

We can also print out the squared values with a while loop and catching the **StopIteration** exception

```
g = squareGenerator()
while(True):
    try:
        print(next(g))
    except StopIteration:
        break
```

1
2
4
9
16
25
36
49
64
81
100

We can also print out the values from the iterator result in a for loop.

```
for x in g:
    print(x)
```

You can retrieve all the values at once from the squareGenerator by enclosing the generator in a list.

```
g = squareGenerator(10)
x = list(g)
print(x)
```

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Sending a value to a Generator

We can send a value the generator yield method using the **send** method. This comes in handy then you want to change the value inside the generator when it is running.

The **send** method sends a value to the generators **yield** method

```
g.send(8)
```

the yield method return a value

```
val = (yield i)
```

if the value is not None then we set the generator value i to val

Here is a counter generator that will receive values from a send method, and update the counter generator count value.

```
def counterGenerator (n):
```

```
    i = 0
```

```
    while i < n:
```

```
        val = (yield i)
```

```
        # If value provided, change counter
```

```
        if val is not None:
```

```
            i = val
```

```
        # else increment value
```

```
    else:
```

```
        i += 1
```

Here we make a counter generator to count from 0 to 9. when the counterGenerator receives a value from the send method its count value changes.

We use the next(g) method call to retrieve values from the generator. We use the **g.send** method to send values to the generator

```
g = counterGenerator(10)
```

```
try:
```

```
    print (next(g)) # 0  
    print (next(g)) # 1  
    print (g.send(8)) # 8  
    print (next(g)) # 9  
    print (next(g)) # 10
```

```
except StopIteration:
```

```
    pass
```

The syntax for python version 2.7 is much different

```
# for python version 2.7
```

```
def counterGenerator (n):
```

```
    i = 0  
    while i < n:  
        val = (yield i)  
        # If value provided, change counter  
        if val is not None:  
            i = val  
        # else increment value  
        else:  
            i += 1
```

```
try:
```

```
    g = counterGenerator(10)  
    print (g.next()) # 0  
    print (g.next()) # 1  
    print (g.send(8)) # 8  
    print (g.next()) # 9  
    print (g.next()) # 10
```

```
except StopIteration:
```

```
    pass
```

LESSON 8 HOMEWORK Part3

Question 3

Make a **FactorialGenerator** that out put all the !factorials of a number 1 to n;

Example factorial !5 = $5*4*3*2*1 = 120$

Set n = 10.

Send a value to the generator after it reaches 5 to start over again.

Zip

Zip allows you the compress values together from separate lists into 1 list,

```
numbers = [1, 2, 3]
```

```
letters = ['a', 'b', 'c']
```

```
zipped = zip(numbers, letters)
```

A iterator of tuples is returned. We can enclose the returned iterator in a list to retrieve the tuples as a list.

```
zlist = list(zipped)
```

```
print(zlist)
```

```
[(1, 'a'), (2, 'b'), (3, 'c')]
```

From the zipped list we can print out the tuples on separate lines:

```
for x in zlist:
```

```
    print(x)
```

```
(1, 'a')  
(2, 'b')  
(3, 'c')
```

We can even iterate through the zipped list , and do some operation on the values inside the tuple. (each letter b is replicated by a)

```
for a,b in zlist:  
    print (b * a)
```

```
a  
bb  
ccc
```

Unzipping a ZIP

We can use List Comprehension to unzip a list:

Our zipped list from above is:

```
zlist = [(1, 'a'), (2, 'b'), (3, 'c')]  
list1 = [ i for i,j in zlist]  
list2 = [ j for i,j in zlist]
```

```
print(list1) # [1, 2, 3]  
print(list2) # ['a', 'b', 'c']
```

We can even use zip to unzip a list

Here is the zlist again

```
zlist = [(1, 'a'), (2, 'b'), (3, 'c')]
```

We first use the *operator on *zlist to separate the zlist

```
[(1, 'a'), (2, 'b'), (3, 'c')]
```

into separate arguments

```
(1, 'a') (2, 'b') (3, 'c')
```

We then use zip on the *zlist:

```
unzipped = zip(*zlist)
```

We print the unzipped object in a list

```
print(list(unzipped))
```

```
[ (1,2,3) , ('a', 'b', 'c') ]
```

Lesson 8 Homework Part4

Question 4

Make a 2 lists of numbers 1 to 5

Zip the two number lists together

Use List Comprehension to multiply the individual numbers together into another list, that just printout the odd numbers.

Example:

```
List1 = [1,2,3,4,5]
```

```
list2 = [1,2,3,4,5]
```

The result should be list 3:

```
list3 = [1,9,15,25]
```

Higher Order Functions

Higher order functions use functions as arguments. These functions are used to do operations on each element of list that is also passed t it.

map function

The map function receives a function and a list. The map applies the function to each item in a list. The map returns an iterator of values that is the result of applying the function to each item in the list.

```
iterator = map(function, list)
```

A simple example is adding 1 to each element in a list.

We first make a list of numbers

```
list1 = [1,2,3,4,5]
```

We then define a function to add 1 to its input parameter x

```
def f(x):
```

```
    return x + 1
```

We then use the map function to apply the function f to each element in the list

```
itr = map(f,list1)
```

The map function returns an iterator of the result of adding 1 to each element in the list.

We need to convert an iterator into a list for printout

```
list2 = list(itr)
```

```
print(list2) # [2, 3, 4, 5, 6]
```

each element in the list now has 1 added to it

The above code is similar to:

```
list1 = [1, 2, 3, 4, 5]
```

```
list2 = []
```

```
for i in list1:
```

```
    list2.append(f(i))
```


to do:

Make a square function and use the map to square the numbers in a list

using an anonymous function lambda

An **anonymous function** is a inline function that has no defined name and can be used within another programming statement to calculate a value on the fly. Since the function does not need a name it is given the anonymous name **lambda**. A lambda function is very convenient.

lambda parameters(s) : statement(s)

A anonymous function to add 1 to x would be

```
lambda x: x+1
```

x is the input parameter and x+1 is a programming statement that returns a value which is equivalent to:

```
def f(x):
```

```
    return x + 1
```

we can use our anonymous function in a map function as follows

```
itr = map(lambda x: x+1, list1)
```

this is quite convenient!

Our complete code using lambda is now:

```
list1 = [1, 2, 3, 4, 5]
```

```
itr = map(lambda x: x+1, list1)
```

```
print(list(itr)) # [2, 3, 4, 5, 6]
```

to do:

Make a lambda square function and use the map function to square the numbers in a list.

Using lambda function's

We can equate a variable to a one parameter lambda function like this:

```
f = lambda x: x*x
```

we can call the lambda function like this:

```
x = f(5)
```

we print out the result like this:

```
print(x)
```

```
25
```

lambda function with more than 1 parameters

```
f = lambda x,y: x + y
```

```
x = f(2,3)
```

```
print (x)
```

```
5
```

lambda function with no parameters

```
f = lambda : 5
```

```
x = f ()
```

```
print(x)
```

```
5
```

Higher order lambda functions

Make function 1

```
f = lambda x: x*x
```

make lambda function 2 that takes lambda function 1

```
f2 = lambda x: f(x) + x
```

call function f2 with function 1, function 1 receives a 5

```
x = f2(f(5))
```

print out results

```
print(x)
```

650

which is:

$f(5) = 25$

$f2(f(5)) = f(25) + 25 = 625 + 25 = 650$

to do

Try your own higher order lambda function f and f2 like cubing squaring and/or adding etc

Lesson 8 Homework Part 5

Question 5

Make a list of numbers 1 to 5. Using a **map** and a lambda inline function square the numbers of the list. Feed the results to a List Comprehension that will cube the even squared numbers. Print the list before and after.

You should get something like this:

List of numbers 1 to 5: [1, 2, 3, 4, 5]

Squared numbers: [1,4,9,16,25]

Cubed of even squared numbers: [64, 4096]

Filter function

The filter function operates similar to map but returns an iterator of values that meet a certain condition like selecting, even or odd numbers.

```
iterator = filter(function, list)
```

We first make a function called **even** that returns true if a number is even

```
def even(x):  
    return x % 2 == 0
```

we then apply the filter function

```
list1 = [1, 2, 3, 4, 5, 6]  
itr = filter(even,list1)  
list2 = list(itr)  
print(list2) # 2 4 6
```

notice only the even numbers are printed out

For convenience we can also use a lambda anonymous function

```
list1 = [1, 2, 3, 4, 5, 6]  
itr = filter(lambda x: x % 2 == 0,list1)  
list2 = list(itr)  
print(list2)
```

TODO

Make a odd function and use the filter function to print out the odd numbers.

Use the filter function and a lambda function to print out the odd numbers.

LESSON 8 HOMEWORK Part6

Question 6

Using List Comprehension or a Generator make a list of numbers 1 to 10.

Use a map, a filter and a lambda function to print out all the odd squared numbers.

You should get something like this:

List of numbers 1 to 10: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

List of odd squared numbers: [1, 9, 25, 49, 81]

Reduce

Reduce takes the first 2 numbers of a list and applies a function to them like adding or subtracting and then applies the function to the **result** and the **next** number and return the result.

```
list = reduce(function,list)
```

The function parameter takes 2 arguments (x,y)

We make a function that adds 2 numbers

```
def add(x,y):  
    return x+y
```

We can now use our add function with **reduce** on a list

You need to import **functools** to use **reduce**

```
from functools import reduce  
list1 = [1,2,3,4,5]  
result = reduce(add, list1)  
print(result) # 15 = 1 + 2 + 3 + 4 + 5
```

reduce works like this

```
x = 0
list1 = [1,2,3,4,5]
for y in list1:
    x = x + y
print(x) #15
```

Basically you adding each number in the list [1,2,3,4,5] to the result

```
1 + 2 = 3
3 + 3 = 6
6 + 4 = 10
10 + 5 = 15
```

We can also call `reduce` with a `lambda`

```
list1 = [1, 2, 3, 4, 5]
result = reduce(lambda x, y: x + y, list1)
print(result) #15
```

You can also use functions that compare values. It starts at the first number and compares the left number to the right number till it gets to the last number.

We first make a less, greater or equal compare function

```
def greater(x,y):
    return x > y
```

We can now test if all the numbers are increasing in the list

```
list1 = [1, 2, 3, 4, 5]
result = reduce(greater, list1)
print(list2) # False
```

For convenience we can replace our `greater` function with a `lambda` inline function. Using the `lambda` function can now test if all the numbers are increasing in the list.

```
list1 = [1, 2, 3, 4, 5]
result = reduce(lambda x, y: x > y, list1)
print(result) # False
```

Using the lambda function we can now test if all the numbers are decreasing in the list .

```
result = reduce(lambda x, y: x < y, [1, 2, 3, 4, 5])
print(result) # True
```

to do:

Apply the reduce function on a function that multiplies 2 numbers together and compares if it is greater than adding them together.

LESSON 8 HOMEWORK Part 7

Question 7

Make a **lambda** in line function that uses **list comprehension** to produce a sequence of numbers 1 to n that has an input variable n. Using the lambda inline function pass 5 to n, then feed the numbers to a **map** and a function to square them using another lambda inline function. Then use a **filter** to return the even squared numbers using another lambda inline functions and then use **reduce** to add them together using another lambda inline function..

You should have results like this:

Numbers 1 to n using lambda function passing 5 to it: [1, 2, 3, 4, 5]

Numbers squared using map : [1, 4, 9, 16, 25]

Even squared numbers using filter : [4, 16]

Result of adding squared numbers using reduce: 20

Bonus: Try filtering out the odd square numbers instead.

Put all your homework into a file called homework8.py

LESSON 9 RECURSION

When a function calls itself it is known as **recursion**. Recursion is analogous to a while loop. Most while loop statements can be converted to recursion, most recursion can also be converted back to a while loop.

The simplest recursion is a function calling itself printing out a message.

```
def print_message():  
    print("I like programming\n");  
    print_message();
```

Unfortunately this program will run forever.

```
I like programming  
I like programming  
I like programming  
I like programming  
I like programming  
...
```

We can add a counter **n** to it so it can terminate at some point.

```
def print_message(n):  
    if(n > 0):  
        print("I like programming");  
        print_message(n-1)
```

```
I like programming  
I like programming  
I like programming  
I like programming  
I like programming
```

You should now run the recursion function

You would call the function like this:

```
print_message(5);
```

It will print I like programming 5 times.

Every time the print_message function is called n decrements by 1

When n is 0 the recursion stops. You may place the statement `print("I like programming")` before or after the recursive call. If you put it before than the message is printed first before each recursive call.

If you put after than the message is printed after all the recursive calls are made. This is quite a difference in program execution.

The operation is very similar to the following while loop:

```
n = 5  
while(n > 0):  
    print("I like programming")  
    n -= 1
```

Recursion is quite powerful, a few lines of code can do so much.

n start's at 5 and decreases by 1 each time the `print_message` function is called: For each recursive call each individual value of n is saved.

5
4
3
2
1

when n becomes 0 then recursion stops.

0

Then the function unwinds and the stored n values get restored in reverse order

1
2
3
4
5

For the next example we will count all numbers between 1 and n. This example may be more difficult to understand, since recursion seems to work like magic, and operation runs in invisible to the programmer.

```
def countn(n):  
    if n == 0:  
        return 0  
    else:  
        return countn(n-1) + 1
```

You call **countn** with a number like this:

```
x = countn(5)      # would return 5 because 1 + 1+ 1+ 1+ 1 = 5  
print(x)          # 5
```

When (n == 0) this is known as the base case. When n == 0 the recursion stops and 0 is return to the last recursive call. Otherwise the **countn** function is called and n is decremented by 1

It works like this:

```
main calls countn(5) with n = 5  
countn(5) calls countn(4) with n=4  
countn(4) calls countn(3) with n=3  
countn(3) calls countn(2) with n = 2  
countn(2) calls countn(1) with n = 1  
countn(1) calls countn(0) with n = 0  
countn(0) returns 0 to count(1) since n == 0  
countn(1) adds 1 to the return value 0 and then returns 1 to count(2)  
countn(2) adds 1 to the return value 1 and then returns 2 to count(3)  
countn(3) adds 1 to the return value 2 and then returns 3 to count(4)  
countn(4) adds 1 to the return value 3 and then returns 4 to count(5)  
countn(5) adds 1 to the return value 4 and then returns 5 to main()  
main() receives 5 from count(5) and prints out 5
```

The statement **return countn(n-1) + 1** is used to call the function recursively and also acts as a place holder for the value returned by the called function.

We could rewrite the recursive part as follows:

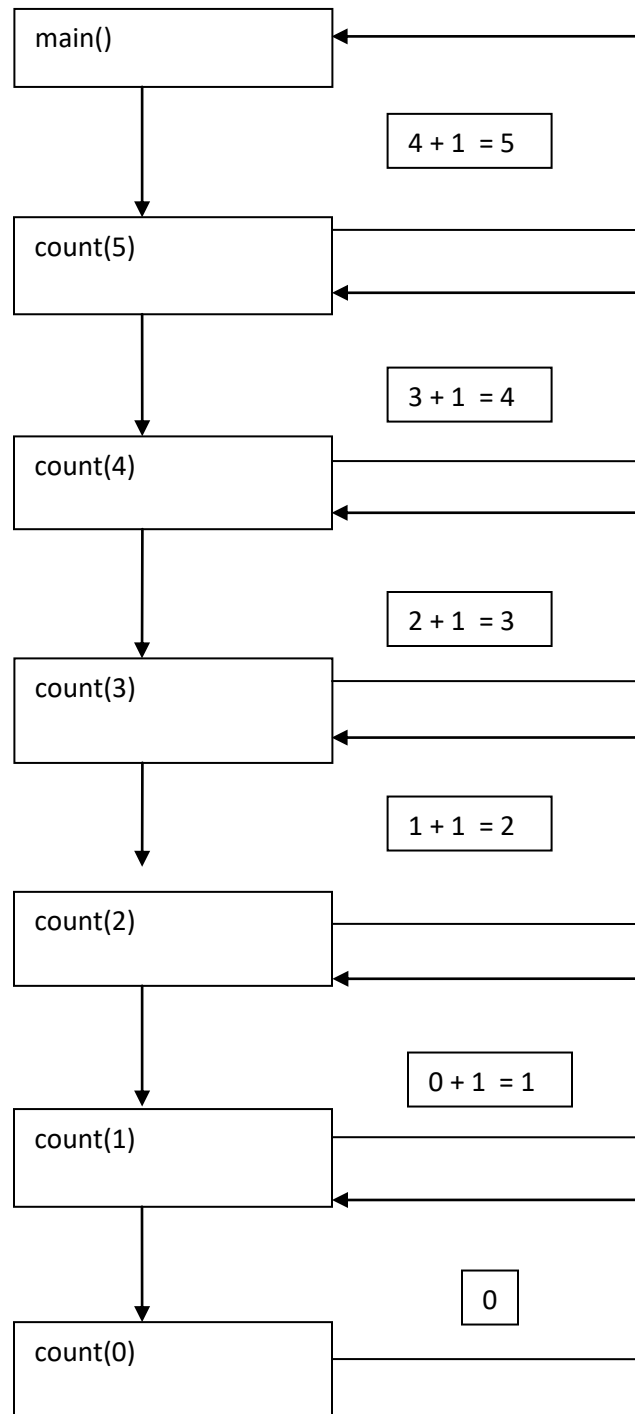
```
x = countn(n-1)
```

```
return x + 1
```

x will now receive the return value from the function call and 1 will be added to the return value and this new value will be returned to the caller.

If you can understand the above then you understand recursion

If you cannot then maybe the following diagram will help you understand.



You probably don't need to understand how recursion works right away. Sometime you just need to accept things for now then understand later. One day it will hit you when you are thinking about something else.

Basically recursion works like this:

For every recursive function call the parameter and local variables are stored. Technically they are stored in temporary memory called a stack.

Every time the function returns the previous numbers that were stored are restored and now become the current number, to be used to do a calculation. The numbers are restored in **reverse** order.

Function call/ return	N
call count(n-1)	5
call count(n-1)	4
call count(n-1)	3
call count(n-1)	2
call count(n-1)	1
count(n-1) returns 0	0
count(n-1) returns 0 + 1	1
count(n-1) returns 1 + 1	2
count(n-1) returns 2 + 1	3
count(n-1) returns 3 + 1	4
count(n-1) returns 4 + 1	5

The thing to remember about recursion is it always return's back where it is called. Here are some more recursive function examples:

Sum numbers 1 to n

We sum numbers from 1 to n.

```
def sumn(n):  
    if n==0:  
        return 0  
    else:  
        return sumn(n-1) + n
```

You would call **sumn** like this:

```
x = sumn(5) # would return 15
print(x)    # print 15
```

It works similar to **countn** instead of adding 1 its adds n.

$$0+1+2+3+4+5 = 15$$

Our counter n serves 2 purposes a recursive counter and a number to add.

Multiply numbers 1 to n (factorial n)

We can also make a **multn** function which multiplies n rather than adding n. This is basically factorial n.

```
def multn(n):
    if n==0:
        return 1
    else:
        return multn(n-1) * n
```

You would call **multn** like this

```
x = multn(5) # would return 120
print(x)     # print 120
```

It works similar to **addn** instead of adding n it multiplies n.

$$1*1*2*3*4*5 = 120$$

Our base case returns 1 rather than 0 or else our result would be 0;

Power x^n

Another example is to calculate the power of a number x^n

In this case we need a base parameter b and an exponent parameter n.

```

def pown(b, n):
    if n == 0 :
        return 1
    else
        return pown(b,n-1) * b

```

You would call **pown** like this:

```

x = pown(2,3) # would return 8 because 2*2*2= 8 since 23=8
print(x)      # 8

```

Every time a recursive call is made the program stores the local variables in a call stack. Every time recursive call finishes executing, the save local variables disappear and the previous local variables are available. These are the ones present before the recursive function was called. These save variables may now be used in the present calculations.

For the above example $2^3=8$ the call stack would look like this.

			n=0						
			b=2		1				
			n=1		n=1				
			b=2		b=2		2		
	n=4		n=2		n=2		n=2		
	b=2		b=2		b=2		b=2	4	
n=5	n=5		n=3		n=3		n=3	n=3	
b=2	b=2		b=2		b=2		b=2	b=2	8

Every time the recursive function finished executing it returns a value. Each returning value is multiplied by the base b. In the above case the returning values are 1,2,4 and 8

The return value is the value from the previous function multiplied by b (2)

```

return pown(b,n-1) * b;

```

the function first returns 1 then $1 * b = 1 * 2 = 2$ then $2 * 2 = 4$ and finally $4 * 2 = 8$

efficient power x^n

A more efficient version of pown can be made relying on the fact then even n can return $b * b$ rather than just return $* b$ for odd n

```
def pown2(b,n):  
    if (n == 0):  
        return 1;  
  
    if (n %2 == 0):  
        return pown2(b, n-2) * b * b  
  
    else:  
        return pown2(b, n-1) * b
```

You would call `pownn` like this:

```
x = pown2(2,3) # returns 8  
print(x) # 8
```

Operation is now much more efficient $1 * 2 * 4 = 8$

Summing a sequence

Adding up all the numbers in a sequence $n * (n + 1) / 2$

n $n * (n + 2) / 2$

0	0
1	1
2	4
3	9
4	12
5	15

Total: 42


```

def seqn(n):
    if n == 0:
        return 1
    else:
        return (n * (n + 2))// 2 + seqn(n-1)

```

You would call `seqn` like this:

```
x = seqn(5) # returns 42
```

```
print(x) #42
```

You should print out the individual values of the sequence for `n` as it calculates the sum of the sequences

```

x = (n * (n + 2))// 2
print(x)
return x + seqn(n-1)

```

Fibonacci sequence

Recursion is ideal to directly execute recurrence relations like Fibonacci sequence. The Fibonacci numbers are the numbers in the following integer sequence.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

In mathematical terms, the sequence f_n of Fibonacci numbers is defined by the recurrence relation.

$$f_n = f_{n-1} + f_{n-2}$$

with seed values

$$f_0 = 0 \text{ and } f_1 = 1$$

A **recurrence relation** is an **equation** that defines a sequence based on a rule that gives the next term as a function of the previous term(s).

def fib(n):

if n == 0:

return 0

elif n == 1:

return 1

else:

return fib(n-1) + fib(n-2)

You would call **fib** like this:

x = fib(10) # would return 55

print(x) # 55

Notice The recursive statement is identical to the recurrence relation

Combinations

We can also calculate combinations using recursion.

Combinations are how many ways can you pick **r** items from a set of **n** distinct elements.

nCr means **n choose r**

n is the number of elements in the set

r is the items you want to choose from the set

Example:

Pick two letters from set $S = \{A, B, C, D, E\}$

Answer is:

$\{A, B\}, \{B, C\}, \{B, D\}, \{B, E\}, \{A, C\}, \{A, D\}, \{A, E\}, \{C, D\}, \{C, E\}, \{D, E\}$

There are 10 ways to choose 2 letters from a set of 5 letters. The combination formula is

$${}^n C r = \frac{n!}{r! (n - r)!}$$

The Recurrence relation for calculated combinations is:

base cases:

$${}^n C n = 1 \quad \text{where} \quad r = n$$

$${}^n C 0 = 1 \quad \text{where} \quad r = 0$$

recursive case:

$${}^n C r = {}^{n-1} C r + {}^{n-1} C r-1 \quad \text{for } n > r > 0$$

Our recursive function for calculating combinations, based on the recurrence relation is:

def combinations(n, r):

if r == 0 or n == r:

return 1

else:

return combinations(n-1, r) + combinations(n-1, r-1)

You would run **combinations** like this:

x = combinations(5,2) # returns 10

print(x) # 10

Print a string out backwards

With recursion printing out a string backwards is easy, it all depends where you put the print statement. If you put before the recursive call then the function prints out the characters in reverse. Since n goes from n-1 to 0. If you put the print statement after the recursive call then the characters are printed not reverse since n goes from 0 to n.

reverse a string

This function prints out a string in reverse.

```
def print_reverse(s,n):
```

```
    if n == 0:
```

```
        print(s[0])
```

```
    else:
```

```
        print(s[n],end="")
```

```
        print_reverse(s, n-1)
```

You would run `print_reverse` like this:

```
s = "tomorrow"
```

```
print_reverse(s,len(s)-1)
```



worromot

Check if a string is a palindrome

A palindrome is a word that is spelled the same forward as well as backwards: Like "radar" and "racecar"

We make a pointer *i* to the start of the word and a pointer *j* to the end of the word.

```
  r a d a r
  ^     ^
  i     j
```

We then move the i forward and the j backward until they meet.

```
  r a d a r
    ^  ^
    i  j

  r a d a r
    ^  ^
    i  j
```

If pointer i and j meet then the word is a palindrome, else it is not.

return True if string is a palindrome otherwise return False

def is_palindrome(s, i, j):

if i >= j:

return True

else:

if s[i] != s[j]:

return False

else:

return is_palindrome(s,i+1, j-1)

You would call the is_palindrome function like this:

s = "radar"

x = is_palindrome(s, 0,len(s)-1) # return True

print(x) # True

s= "apple"

x =is_palindrome(s, 0,len(s)-1) # return False

print(x) # False

Permutations

Permutations are how many ways you can rearrange a group of numbers or letters. For example for the string “ABC” the letters can be rearranges as follows:

ABC
ACB
BAC
BCA
CBA
CAB

Basically we are swapping character and then print them out
We start with ABC if we swap B and C we end up with ACB

```
# print permutations of string s  
def print_permutations(s, i, j):  
    # print out permutation  
    if i == j:  
        print(s)  
    else:  
        for k in range(i, j+1):  
            # swap i and k  
            c = s[i]  
            # s[i] = s[k]  
            s = s[:i] + s[k] + s[i+1:]  
            #s[k] = c  
            s = s[:k] + c + s[k+1:]  
  
            # recursive call  
            print_permutations(s, i + 1, j)  
            # put back, swap i and k  
            c = s[i]  
            #s[i] = s[k]  
            s = s[:i] + s[k] + s[i+1:]  
            #s[k] = c;  
            s = s[:k] + c + s[k+1:]
```

You would call the `print_permutations` function like this:

```
s = "ABC";  
print_permutations(s, 0, len(s)-1);
```

ABC
ACB
BAC
BCA
CBA
CAB

to do:

try different strings

Combination sets

We have looked at combinations previously where we wrote a function to calculate how many ways you can choose r letters from a set of n letters.

nCr n choose r

Combinations allow you to pick r letters from set $S = \{A, B, C, D, E\}$

$n = 5$ $r = 2$ $nCr = 5C2$

Answer:

$\{A, B\}, \{B, C\}, \{B, D\}, \{B, E\}, \{A, C\}, \{A, D\}, \{A, E\}, \{C, D\}, \{C, E\}, \{D, E\}$

We are basically filling a second character array with all possible letters up to r .

Start with ABCDE we would choose AB then AC then AD then AE etc. We use a loop to traverse the letters starting at $n = 0$, and fill the comb string. When $n = r$ we then print out the letters stored in the comb string

combinations

```
def print_combinations(s, combs, start, end, n, r):
```

```
# Current combination is ready to be printed
```

```
    if n == r:
```

```
        for j in range(r+1):
```

```
            print(combs[j], end="")
```

```
        print("")
```

```
    return;
```

```

# replace n with all possible elements.
i = start
while(i <= end and end - i + 1 >= r - n):
    combs = combs[:n] + s[i] + combs[n+1:]
    print_combinations(s, combs, i+1, end, n+1, r)
    i+=1

```

A B
A C
A D
A E
B C
B D
B E
C D
C E
D E

You would call the print_combinations function like this:

```

s = "ABCDE";
combs = " " * (len(s) + 1)
r = 2

# s, combs,start, end, n, r (5 choose 2)
print_combinations(s, combs,0,len(s)-1,0,r)

```

to do

try different strings and r values

Determinant of a matrix using recursion.

In linear algebra, the determinant is a useful value that can be computed from the elements of a square matrix. The determinant of a matrix A is denoted det(A), detA , or |A

In the case of a 2 × 2 matrix, the formula for the determinant is:

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

For a 3 × 3 matrix A, and we want the s formula for its determinant |A| is

$$|A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix}$$

$$= aei + bgf - ceg - bdi - afh$$

Each determinant of a 2×2 matrix in this equation is called a "minor" of the matrix A. The same sort of procedure can be used to find the determinant of a 4×4 matrix, the determinant of a 5×5 matrix, and so forth.

Our code actually follows the above formula, calculating and summing the minors.

```
# calculate determinant of a matrix
```

```
def determinant(matrix, size):
```

```
    sign=1
```

```
    b = [[0]*3 for i in range(3)] # make empty 2d array
```

```
    # base case
```

```
    if size == 1:
```

```
        return (matrix[0][0]);
```

```
else:
```

```
    det=0
```

```
    for c in range(size):
```

```
        m=0
```

```
        n=0
```

```
        for i in range(size):
```

```
            for j in range (size):
```

```
                b[i][j] = 0
```

```
                if i!=0 and j!=c:
```

```
                    b[m][n] = matrix[i][j];
```

```
                    if n< (size-2):
```

```
                        n+=1
```

```
                    else:
```

```
                        n=0
```

```
                        m+=1
```

```
            det = det + sign*(matrix[0][c]*determinant(b,size-1));
```

```
            sign = -1*sign; # toggle sign
```

```
    return (det)
```

You call and run the determinant function like this:

```
m = [[6,1,1],[4,-2,5],[2,8,7]];
```

```
x = determinant(m,3)
```

```
print(x)
```

-306

There are many more recursive examples, too numerous to present. If you do all the following to do questions you will be a recursive expert.

Todo

For questions 1 to 7 use the list: **a = [1,2,3,4,5]**

1. Write a function to print out a list forwards using recursion.
called **printForward(a, n)**:
2. Write a function to add up all numbers in a list using recursion
called **add(a,n)**:
3. Write a function to print out all the even numbers in a list using recursion
called **printEven(a,n)**:
4. Write a function to print out odd numbers in a list using recursion
called **printOdd(a,n)**:
5. Write a function to print out a list backwards using recursion
called **printBackwards(a, n)**:
6. Write a function to return the largest number in an list using recursion
called **largest(a,n)**:
7. Write a function to return the smallest number in an list using recursion
called **smallest(a,n)**:

8. Write a recursive function called **reverse_string(s, n)** that reverses a string in place. The recursive string receives the string and returns the string in reverse. No printing inside the function is allowed.

9. Write a recursive function **search_number(a, n)** that searches for a number in an array and return the index of the number it found otherwise returns -1 if not found.

10. Write a recursive function **search_digit(d, x)** that searches for a number in an number and return True if the number if found otherwise returns False if not found. Divide by 10 will get you the next number, mod 10 will get you the last digit. Remember to do integer division.

Examples:

```
print(search_digit(12346789,7))    # True
print(search_digit(12346789,5))    # False
```

11. Write a recursive function called **sum_digits(d)** that adds up all the digits in a number of any length. The recursive function receives an int number and returns the sum of all the digits. Example: `sum_digits(1234) = 10`

Divide by 10 will get you the next number, mod 10 will get you the last digit. Remember to do integer division.

12. Write a recursive function called **format_number(s, n)** that can insert commas in a string number. For example 1234567890 becomes 1,234,567,890

13. Write a recursive function called **format_string(s)** using slices that can insert commas in a string number. For example 1234567890 becomes 1,234,567,890

14. Write a recursive function **is_even(n)** that return True if a number has even count of digits or false if the number of digits is odd.

15. Write a recursive function **print_binary(d)** that would print a decimal number as a binary number. A binary number just has digits 0 to 1.

Where a decimal number has digits 0 to 9. The decimal number 5 would be 0101 in binary, since $1*1 + 0*2 + 1*4 + 0*8 = 1 + 4 = 5$. We are going left to right.

To convert a decimal number to binary You just need to take mod 2 of a digit and then divide the number by 2

$$5 \% 2 = 1 \rightarrow 1$$

$$5 / 2 = 2$$

$$2 \% 2 = 0 \rightarrow 0$$

$$2 / 2 = 1$$

$$1 \% 2 = 1 \rightarrow 1$$

$$1 / 2 = 0$$

$$0 \% 2 = 0 \rightarrow 0$$

We are done so going backwards

5 in binary is 0 1 0 1

Recursion is good for going backwards.

16. Write a recursive function **is_prime(n)** that returns True if a number is prime otherwise False.

A prime number can only be divided evenly by itself. 2,3,5,7, are prime numbers. You can use the mod operator % to test if a number can be divided evenly by numbers other than itself. Example: $4 \% 2 = 0$ Because 4 can be divided evenly by another number like 2 so therefore 4 is not a prime number

17. Rewrite **isPalindrome** using substring slices [:] instead rather than using i and j. Try to ignore spaces, and punctuation.

Put all your functions in a python file called Lesson9.py Include a main function that tests all the recursive functions.

18. Make a recursive function called **partition(a,n)** that will partition an array in place into odd and even numbers

Example: `partition([1, 2, 3, 4, 5, 6],0)`

Array before:

[1, 2, 3, 4, 5, 6]

Array After:

[1, 3, 5, 2, 4, 6]

19 Write a recursive function that calls another recursive function

The first function **f1(start, end)**

could calculate factorial from start to end maybe 2 numbers at a time

The second function **f2(n)** would multiply the different ranges of factorial results

f2(5) would work like this

n	start	end	result	total
5	5	4	20	20
3	3	2	6	120
1	1	0	1	120

print(f2(5)) # 120

20 Change function f2 from Question 19 to receive the f1 function as a parameter

fn(f1,n)

The result should be the same

print(fn(f1,5)) # 120

Lesson 10 Regular Expressions

Regular expressions let you search for string patterns in a text string. Regular expressions are a little difficult to understand and use but once you realize they are just using letters to specify a pattern that you can match.

The simplest regular expression is a string of letters like "are" that you can use to determine if a text string contains this word pattern.

Example: "Happy days are here again" contains the word pattern "are".

Python has function's used to locate patterns in a string, contained in the **re** module

You would import as follows:

```
import re
```

re module Regular Expression functions:

Function	Description	Example Using
findall	Returns a list containing all matches in the string str	re.findall("are", str)
search	Returns a Match object if there is a match anywhere in the string str	re.search("are", str)
match	Returns a Match object if there is a match at the beginning of the string str	re.search("are", str)
split	Returns a list where the string str has been split at each match	re.split(",", str)
sub	Replaces one or many matches with a string in the string str	re.sub("always", "forever", str)

Each regular expression returns a match object. A Match Object is an object containing information about the search and the result.

Note: If there is no match, the value None will be returned, instead of the Match Object.

Match object function methods and properties

<code>span()</code>	returns a tuple containing the start-, and end positions of the match.
<code>.string</code>	returns the string passed into the function
<code>.group()</code>	returns the part of the string where there was a match

Using search

The **search()** function searches the string for a match, and returns the above Match object if there is a match at the beginning of the string str.

Example using:

```
str = "I like Python"  
m = re.search("like",str)  
print(m.span())  
print(m.string)  
print(m.group())
```

```
(2, 6)  
I like Python  
like
```

We have search for the pattern "like" in the text string "I like Python" where the match object has found the pattern in columns 2 to 6

span =(2, 6) match found in columns 2 to 6

string = the sting to be searched: "I like Python"

group = we only have 1 group result: "like"

To do:

search for something that you can find and not found in the string "I like Python"

Using match

The **match()** function searches the string for a match, and returns the above Match object if there is a match.

If there is more than one match, only the first occurrence of the match will be returned:

Example using:

```
s = "I like Python"  
m = re.match("like",s)  
print(m.span())  
print(m.string)  
print(m.group())
```

```
(2, 6)  
I like Python  
like
```

We have search for the pattern "like" in the text string "I like Python" where the match object has found the pattern in columns 2 to 6

Using findall

The **findall()** function returns a list containing all matches found in a string.

Example using:

```
s = "I like Python always"  
m = re.findall("y",s)  
print(m)
```

```
['y', 'y']
```

The findall has found 2 y's in our string "I like Python always". It is too bad it does not tell us the words containing the y's and where the word is located in the string.

To do: try using a word rather than a letter

Using split

The split() function returns a list where the string has been split at each match:

```
s = "I like Python always"  
m = re.split(" ",s)  
print(m)
```

```
['I', 'like', 'Python', 'always']
```

todo: try splitting on a letter or a comma

Using sub

The **sub()** function replaces a string with the another string of your choice:

```
s = "I like Python always"  
m = re.sub("always", "forever", s)  
print(m)
```

```
I like Python forever
```

We have replaced the word “always” to “forever”

Regular expression have **sequences**, **metacharacters** and **sets** to make regular expressions more powerful.

A **special sequence** is a \ followed a characters and has a special meaning, like \s to represent white space.

Metacharacters are characters with a special meaning, like + which means 1 or more matches.

A **set** is a set of characters inside a pair of square brackets [] with a special meaning, where [A-Z] matches any letter A to Z.

Here are the tables of Sequence, metacharacters and sets:

Sequences

A special sequence is a \ followed by one of the characters in the list below, and has a special meaning:

Sequence	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"

\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Python\Z"

Metacharacters

Metacharacters are characters with a special meaning that allows the metacharacter to represent many other characters. Example the dot . can represent any character.

Metacharacter	Description	Example
[]	A set of characters from start to end separated by a -	"[a-m]"
\	Signals a special sequence (can also be used to escape special characters like \()	"\d"
.	Any character (except newline character)	"."
^	Starts with	"^happy"
\$	Ends with	"days\$"
*	Zero or more occurrences	"a*"
+	One or more occurrences	"a+"
?	Zero or more occurrence	a?"
{ }	Exactly the specified number of occurrences	"a{2}"
	Either or	"yes no"
()	Capture and group	"(\w+)"

Sets

A set is a set of characters inside a pair of square brackets [] with a special meaning.

Set	Description
[arn]	Returns a match where one of the specified characters (a, r, or n) are present
[a-n]	Returns a match for any lower case character, alphabetically between a and n
[^arn]	Returns a match for any character EXCEPT a, r, and n
[0123]	Returns a match where any of the specified digits (0, 1, 2, or 3) are present
[0-9]	Returns a match for any digit between 0 and 9
[0-5][0-9]	Returns a match for any two-digit numbers from 00 and 59

[a-zA-Z]	Returns a match for any character alphabetically between a and z, lower case OR upper case
[+]	In sets, +, *, ., , (), \$, {} has no special meaning, so [+] means: return a match for any + character in the string

Using metacharacters, sets and sequences

If you want to match a digit use `\d`

If you want to match a UPPER CASE letter use `[A-Z]`

If you want to match a lowercase letter use `[a-z]`

If you want to match UPPER and lower case letter use `[a-zA-Z]`

If you want to match a space use `\s`

To match 1 or more spaces `\s+`

To match 3 digits `\d{3}`

To match one or more letters `[a-zA-Z]+`

To match letters and numbers `\w`

To match 1 or many letters and numbers `\w+`

To match 0 or many letters and numbers `\w*`

To match 0 or 1 letters and numbers `\w?`

To make a group `()`

To match any letter `.`

Validating text strings using match function

The **match()** function of **re** module in Python will search the regular expression pattern and return the first occurrence. The Python RegEx Match method checks for a match only at the beginning of the string. So, if a match is found in the first line, it returns the match object. But if a match is found in some other line, the Python RegEx Match function returns None.

Examples:

us zip code

use `d{5}` to match 5 digits

```
m = re.match("\d{5}", "12345")
print(m)
```

```
m = re.match("\d{5}", "1234")
print(m)    # none
```

to do: try numbers and letters

Canadian postal code

use `[a-zA-Z]` to represent all upper and lower case letters

```
m = re.match("[a-zA-Z]\d[a-zA-Z]\d[a-zA-Z]\d", "M2J2Y5")
```

to do:

(1) try out your own postal code

(2) put a space in your postal code and then adjust the regular expression to accommodate the space

Phone number

(123) 456-7890

A phone number has 3 digits surrounded by round brackets: (123)

So we use `\d` to represent digits 0-9 and then use `{3}` for three digits contained in round brackets. **`(\d{3})`**

The round brackets are metacharacters that have to be escaped with forward slashes. **`(\d{3})`**

Continuing we have a space followed by three more digits `\d{3}` a hyphen – followed by 4 more digits `\d{4}`

The following is a regular expression for a phone number.

`\(\d{3}\) \d{3}-\d{4}`

(123)		456	-	7890
<code>\(\d{3}\)</code>		<code>\d{3}-\</code>	-	<code>\d{4}</code>

We only have 1 slight problem, many people will forget to enter a space after the round brackets so we use the metacharacter `?` which means 0 or 1 character.

With a space:

```
m = re.match("\(\d{3}\) ?\d{3}-\d{4}", "(123) 456-7890")
print(m)
```

with out a space:

```
m = re.match("\(\d{3}\) ?\d{3}-\d{4}", "(123)456-7890")
print(m)
```

```
<re.Match object; span=(0, 14), match='(123) 456-7890'>
<re.Match object; span=(0, 13), match='(123)456-7890'>
```

The final regular expression as follows to handle a space or empty space

(123)		456	-	7890
<code>\(\d{3}\)</code>	<code>?</code>	<code>\d{3}</code>	-	<code>\d{4}</code>

To do: try you own phone number

using groups ()

A group allows you to pick out and extract parts of the matching text.

Example: picking parts of the first and last name of a name

```
# groups
s = "Tom, Smith"
m = re.search("(\w+), (\w+)",s)
print(m)
print(m.group())
print(m.group(1))
print(m.group(2))
```

Regular Expression Homework

Question 1

Write the regular expression to validate a web page url like:

`http://www.cstutoring.com`

Question 2

With the regular expression to validate a email and use a group the print out the username and host.

Example:

`students@cstutoring.com`

username: students

host: cstutoring.com

Question 3

Write a regular expression to split a line where the word are separated by commas that can contain one or many spaces. Print out the words in a loop each on a separate line.

hint: use split method

Example line:

```
Happy, days, are, here,again\n"
```

Example output:

```
Happy  
days  
are  
here  
again
```

Question4

Write a regular expression to remove all the new lines from a string

Example:

Before:

```
happy days are here again\n"
```

after:

```
happy days are here again"
```

Question 5

Write python code using regular expression(s) to locate all words with a certain letter and print them out.

Hint #1: use split then match on each word.

Hint #2: use * metacharacter which means 0 or many

LESSON 11 SQL and SQLite

SQL (Structured Query Language) allows you to read and write data from a database. A data base stores information in tables. Tables have rows and columns to represent the data you want to store. The columns store the values. A row contains the data columns. A row is also known as a record.

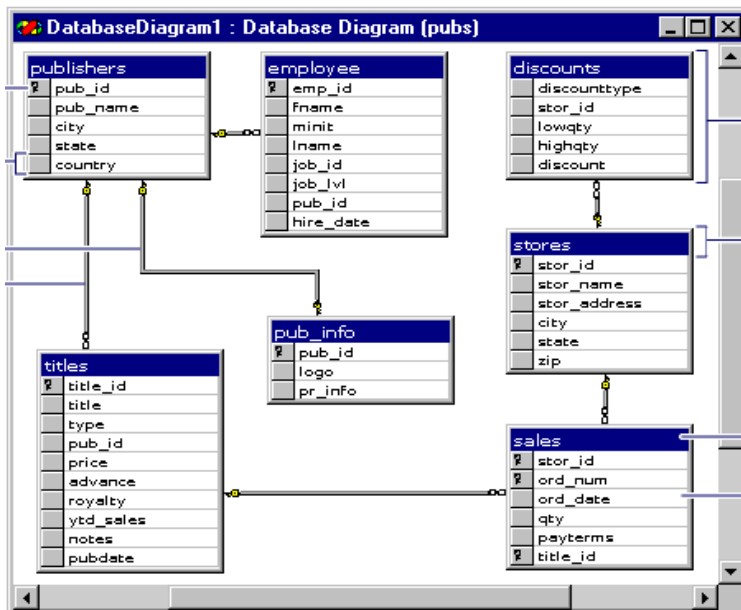
An Example of a Person table would have columns FirstName, LastName, Address, Phone and Email. Each row would contain data for a different person.

You can visualize the Person table like this.

Person Table

FirstName	LastName	Address	Phone	Email
Tom	Smith	42 Ocean Ave	876-9876	tom@mail.com
Sue	Jones	1 One St	645-8654	sue@mail.com
Mary	Berry	7 Hanover St	454-5432	mary@mail.com

A database stores many tables. Tables can represent many things. A data base may have tables to represent a Store. A store would have tables for customers, inventory, orders, etc.



Python makes database programming easy for you. Python has the **SQLite** database built into Python3

The first thing we have to do is to connect to a database.

We first import sqlite3

```
import sqlite3
```

Then we use the sqlite3 **connect** function to connect to our data base.

```
# connect to data base  
conn = sqlite3.connect("lesson11.db")
```

If the data base does not exist it will be made automatically. The data base is just a binary file with the same name you gave it. It will be stored in the same folder where your python file's are stored: Lesson11.db

Creating database table

The next thing we need to do is to add a table to your data base. The table will contain data about a person. We create our **Person** table, by specifying the column names and data types each column will hold.

We first need to know what kind of data a column can represent. Here are the available **sqlite** column data types.

Python Column Data types:

Data Type	Description	Example
NULL	Represents no value	
INTEGER	signed integer	1234
REAL	floating point value	10.5
TEXT	text string,	"tom Smith"
BLOB	blob of data (large strings)	A very large text message

SQL is a data base language to access a data base. SQL has commands to select insert, update and delete rows into a table

Once we know the data types of the columns we then we can use the SQL **Create statement** to create the following **Person** table.

Person Table

FirstName	LastName	Address	Phone	Email
Tom	Smith	42 Ocean Ave	876-9876	tom@mail.com
Sue	Jones	1 One St	645-8654	sue@mail.com
Mary	Berry	7 Hanover St	454-5432	mary@mail.com

Here is the SQL **Create** statement to define the column names and data types for the person table.

```
# sql create statement
sql = "CREATE TABLE Person"
sql += "(FirstName TEXT, LastName TEXT, Address Text, Phone TEXT, Email TEXT)"
```

If you run the program, more than once then you only want to create the person table only if does not exist to avoid table already created error.

```
sql = "CREATE TABLE IF NOT EXISTS Person"
sql += "(FirstName TEXT, LastName TEXT, Address Text, Phone TEXT, Email TEXT)"
```

Next we make a **cursor** object. The cursor object connects to the database and is responsible to execute SQL statements, and to access the individual records in the data base for fetching and inserting updating and deletion of rows.

```
# create cursor object
cursor = conn.cursor()
```

Lastly we execute the SQL statement and the person table is created,

```
# execute sql statement
cursor.execute(sql)
```

You need to commit the transaction after each execution.

```
conn.commit()
```

Inserting data into a table

The next thing we need to do is to insert data into the person table, using the SQL **Insert** command.

Here is the SQL insert command syntax:

```
INSERT INTO table VALUES (value(s))
```

Our columns are: | FirstName | LastName | Address | Phone | Email |

```
# insert data into Person table
```

```
sql = "INSERT INTO Person VALUES ('Tom', 'Smith', '42 Ocean Ave', '876-9876', 'tom@mail.com')"  
cursor.execute(sql)
```

```
sql = "INSERT INTO Person VALUES ('Sue', 'Jones', '1 One St', '645-8654', 'sue@mail.com')"  
cursor.execute(sql)
```

```
sql = "INSERT INTO Person VALUES ('Mary', 'Berry', '7 Hanover St', '454-5432', 'mary@mail.com')"  
cursor.execute(sql)
```

```
# commit transaction  
conn.commit()
```

Inserting data into a table using a tuple or list

When you insert data from a tuple or a list you must also specify the column names and use a ? to indicate where the column values are to be substituted for.

Insert statement syntax:

```
INSERT INTO (column name(s)) table VALUES (?, ?, ?, ?, ?)
```

```
sql = "INSERT INTO Person (FirstName, LastName, Address, Phone, Email) VALUES (?, ?, ?, ?, ?)"  
values = ('Joe', 'Toe', '5 Apple St', '555-5555', 'joe@mail.com')  
cursor.execute(sql, values)  
conn.commit()
```

reading rows from the database

We use the SQL **Select** statement to select which columns we want to read from the data base.

Select statement syntax:

```
SELECT column_names FROM table_name
```

We can specify each column name that we want to access.

```
sql = "SELECT FirstName, LastName, Address, Phone, Email FROM Person"  
rows = cursor.execute(sql).fetchall()
```

Alternately we can use the wildcard * to select all columns of a particular row.

```
# read rows from data base  
  
Sql = "SELECT * FROM Person"  
rows = cursor.execute(sql).fetchall()
```

When we call the **execute** method from the cursor object and the **fetchall** method, would return a list tuples where each tuple represents 1 row of data by calling. The fetchall method fetches all rows from the Person table.

We can then print out the row from the list of row tuples.

```
# print out rows  
for row in rows:  
    print(row)
```

```
('Tom', 'Smith', '42 Ocean Ave', '876-9876', 'tom@mail.com')  
( 'Sue', 'Jones', '1 One St', '645-8654', 'sue@mail.com')  
( 'Mary', 'Berry', '7 Hanover St', '454-5432', 'mary@mail.com')  
( 'Joe', 'Toe', '5 Apple St', '555-5555', 'joe@mail.com')
```

We need to commit the changes we made to the data base.

```
# commit transaction  
conn.commit()
```

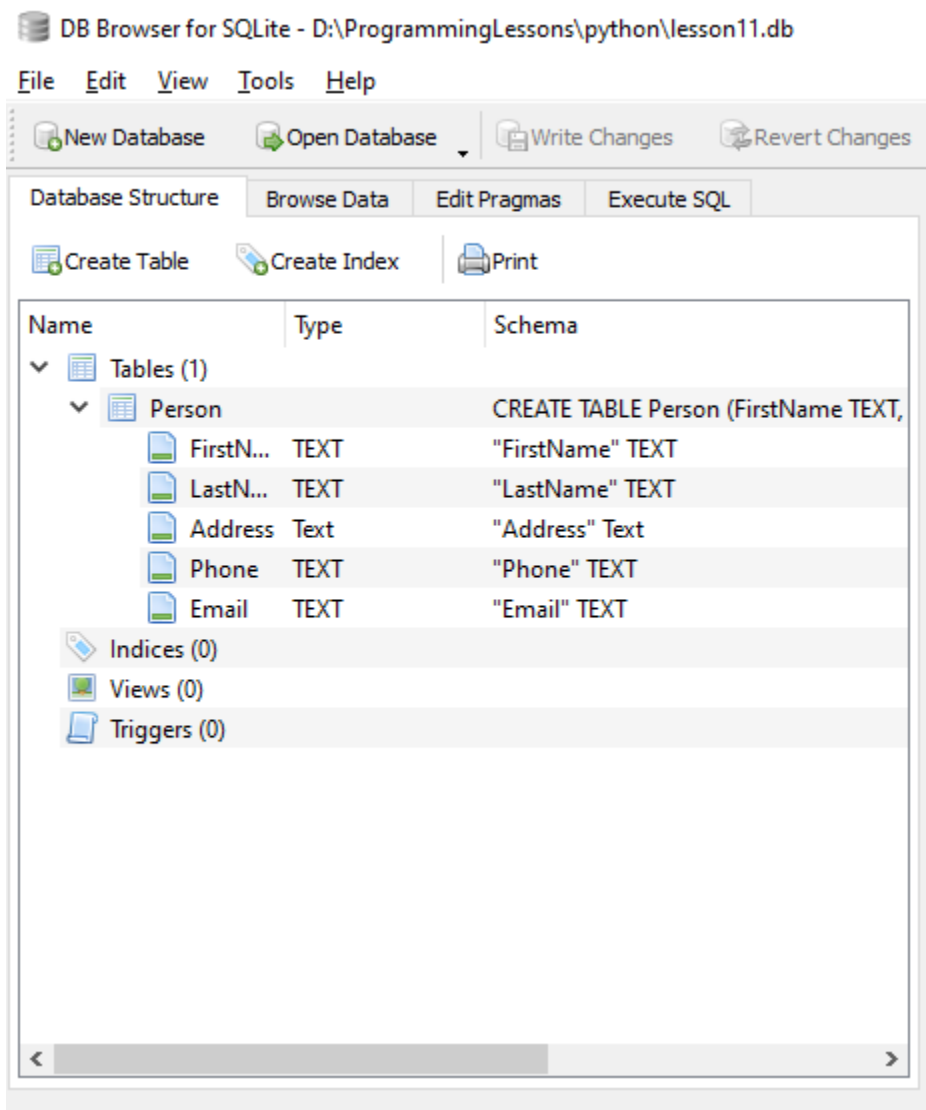
DB Browser for SQLite

DB Browser for SQLite (DB4S) is a high quality, visual, open source tool to create, design, and edit database files compatible with SQLite.

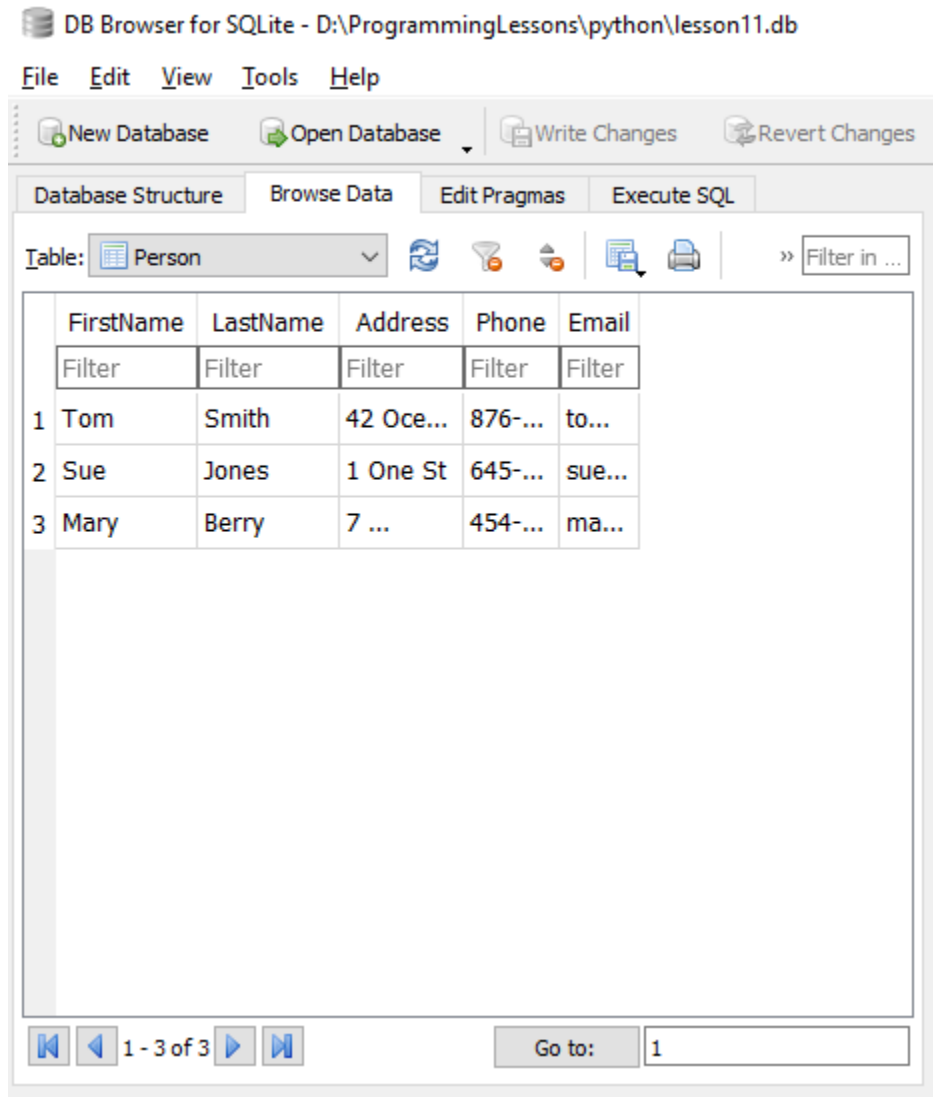
Available from:

<https://sqlitebrowser.org/>

We first display the person table structure



Next we display the person table data



Using Where clause

The **where** clause lets you select specific data row by matching one or many column values.

Syntax of select statement with where clause

Select column(s) from table_name where column=?

The ? is a place holder where the value is substituted from a value that is placed as a tuple in the execute statement. The **column=?** Is known as a condition that is validated true or false.

```
# select where phone = 876-9876
phone = '876-9876'
sql = "SELECT * FROM Person where phone = ?"
rows = cursor.execute(sql,(phone,)).fetchall()

# print out rows
for row in rows:
    print(row)
```

```
('Tom', 'Smith', '42 Ocean Ave', '876-9876', 'tom@mail.com')
```

Note: You need to have an extra comma in the tuple and function call when we have only 1 item.

```
rows = cursor.execute(sql,(phone,)).fetchall()
    ^^
```

Compound where conditions

A where clause can have more than 1 condition using **AND** and **OR** operators.

AND means both conditions must be true to be evaluated as true
OR means either conditions must be true to be evaluated as true

Here is an example using looking for 2 different phone numbers

```
# compound where clause
phone1 = '876-9876'
phone2 = '645-8654'
sql = "SELECT * FROM Person WHERE phone = ? OR phone = ?"
rows = cursor.execute(sql,(phone1,phone2,)).fetchall()

# print out rows
for row in rows:
    print(row)
```

```
('Tom', 'Smith', '42 Ocean Ave', '876-9876', 'tom@mail.com')
('Sue', 'Jones', '1 One St', '645-8654', 'sue@mail.com')
```

Fetching data from a data base

Using the sql Select class we can fetch all rows, the specified number of rows or just one row.

Fetch method	Description
<code>cursor.fetchall()</code>	Fetches all the rows of a query result. It returns all the rows as a list of tuples. An empty list is returned if there is no record to fetch.
<code>cursor.fetchmany(size)</code>	Returns the number of rows specified by size argument. When called repeatedly this method fetches the next set of rows of a query result and returns a list of tuples. If no more rows are available, it returns an empty list.
<code>cursor.fetchone()</code>	Returns a single record or None if no more rows are available

Todo:

Try each of the above fetch methods

Accessing Database data

Getting the number of rows

```
n = len(rows).
```

Accessing a particular row

```
row = rows[0]
```

Accessing a particular column in rows

```
column = rows[0][3]
```

Accessing a particular column in a row

```
column = row[3]
```


UPDATING ROWS

You can update specified column value in a table using the SQL **UPDATE** command.

Here is the update command syntax:

```
UPDATE table SET column = ? WHERE column = ? "
```

Where ? is a place holder that will get substituted to a specified value

```
# update phone number
firstname = 'Tom'
lastname = 'Smith'
phone = '111-2222'
sql = "UPDATE Person SET phone = ? WHERE firstname = ? and lastname = ?"
cursor.execute(sql,(phone, firstname, lastname))

# commit transaction
conn.commit()

# print results
firstname = 'Tom'
lastname = 'Smith'
sql = "SELECT DISTINCT * FROM Person where firstname = ? and lastname = ?"
rows = cursor.execute(sql,(firstname,lastname),).fetchall()

# print out rows
for row in rows:
    print(row)
```

Note: we have used the keyword **DISTINCT** so we only get 1 row selected else we would get more than 1

```
('Tom', 'Smith', '42 Ocean Ave', '111-2222', 'tom@mail.com')
```

Deleting Rows

There will be situations where you will need to remove rows using a specified condition.

To delete a row you use the SQL **DELETE** command.

The syntax is as follows:

```
DELETE FROM table WHERE column = ?
```

Where ? is a place holder that will get substituted to a specified value

```
# delete a row
phone = '645-8654'
sql = "DELETE FROM Person WHERE phone = ?"
cursor.execute(sql,(phone,))

# commit transaction
conn.commit()

# print results
sql = "SELECT * FROM Person"
rows = cursor.execute(sql).fetchall()

# print out rows
for row in rows:
    print(row)
```

```
('Tom', 'Smith', '42 Ocean Ave', '111-2222', 'tom@mail.com')
('Mary', 'Berry', '7 Hanover St', '454-5432', 'mary@mail.com')
```

Dropping Table

There will also be situations where you need to drop a table. You drop a table using the SQL **DROP** command

The syntax is as follows:

```
DROP TABLE table
```

There is an alternate drop table to check if a table exists.

Its syntax as follows

```
DROP TABLE IF EXISTS table
```

```
sql = 'DROP TABLE Person'  
cursor.execute(sql)  
conn.commit()
```

```
sql = 'DROP TABLE IF EXISTS Person'  
cursor.execute(sql)  
conn.commit()
```

Closing Connection

Once you finished using the database you need to close the connection.

```
# close data base  
conn.close()
```

Lesson 11 Homework Question 1

Create a Database that has a Book table

A book will have a Title, Author, Price and year published.

You may want to make a book class that has Title, Author, Price and Year.

The book class has a constructor, getters and setters and a `__str__` method to print out the book details.

The book class can be used to move book data base row data around conveniently.

Make a main program that will create the Book table and has a menu where you can insert, select, update and remove book's using the Book class.

Your Book table may look like this:

Column Name	Description
Title	TEXT
Author	TEXT
Price	REAL
Year	INTEGER

Primary and Foreign keys

A primary key PK is a column that has a unique value for all the rows and is used to identify a particular row using the primary key value. Each table has one and only one primary key. A table may have more than 1 column as the primary key. For example in a the case of a storing a first name and last name. One column for the first name and another column for the second name. In this situation the name is unique.

SQLite allows you to define a primary key in two ways:

First, if the primary key has only one column, you use the **PRIMARY KEY** column constraint to define the primary key as follows:

```
CREATE TABLE table_name(  
    column_1 INTEGER NOT NULL PRIMARY KEY,  
    ...  
);
```

Second, in case primary key consists of two or more columns, you use the **PRIMARY KEY** table constraint to define the primary as shown in the following statement

```
CREATE TABLE table_name(  
    column_1 INTEGER NOT NULL,  
    column_2 INTEGER NOT NULL,  
    ...  
    PRIMARY KEY(column_1, column_2, ...)  
);
```

Foreign key

A Foreign key FK is a key in a table that is a primary key from another table, (like a foreigner in a country from another country). A relation is made between the table that has the primary key to the table that has the foreign key. A foreign key is also a constraint that verifies the existence of the value present in one table to another table.

You specify a foreign key for a primary key in table_name2 as follows

```
CREATE TABLE table_name2
(
  column_1 INTEGER NOT NULL,
  column_2 INTEGER NOT NULL,

  FOREIGN KEY(column2) REFERENCES table_name1 (table_name_primary_key_column name)
);
```

Column2 is the foreign key.

Column 2 has the same name as the primary key in table_name.

Auto increment of Primary Key

Auto increment automatically increments the value of a primary key **id** for each insertion of a record. Primary keys are usually an INTEGER value so it makes auto increment easy. Also auto increment is convenient. The auto increment primary key **id** usually starts a 1 and then get's incremented for each insertion.

The AUTOINCREMENT keyword is used to indicate auto increment.

Here is an example declaring a INTEGER primary key AS AUTOINCREMENT:

```
id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
```

When you insert into a table that uses a auto increment primary key **id** you must not include the AUTOINCREMENT **id** as a value in the insert sql statement. It is supplied for you automatically.

When you use insert statement for auto increment you must supply both the column names and values.

```
sql = "INSERT INTO Items ( Name, Description, Price, Quantity)
VALUES ('Apples', 'Shinny Red',1.95,100)"
```

Primary Key and Foreign Key Example

A Customer has a CustomerID,Name,Address,Email,Phone

An Item has a ItemId, Name,Description,Price,Quantity

An Order has a OrderID,CustomerID and a ItemID

The CustomerID of the Customers table is a Primary Key

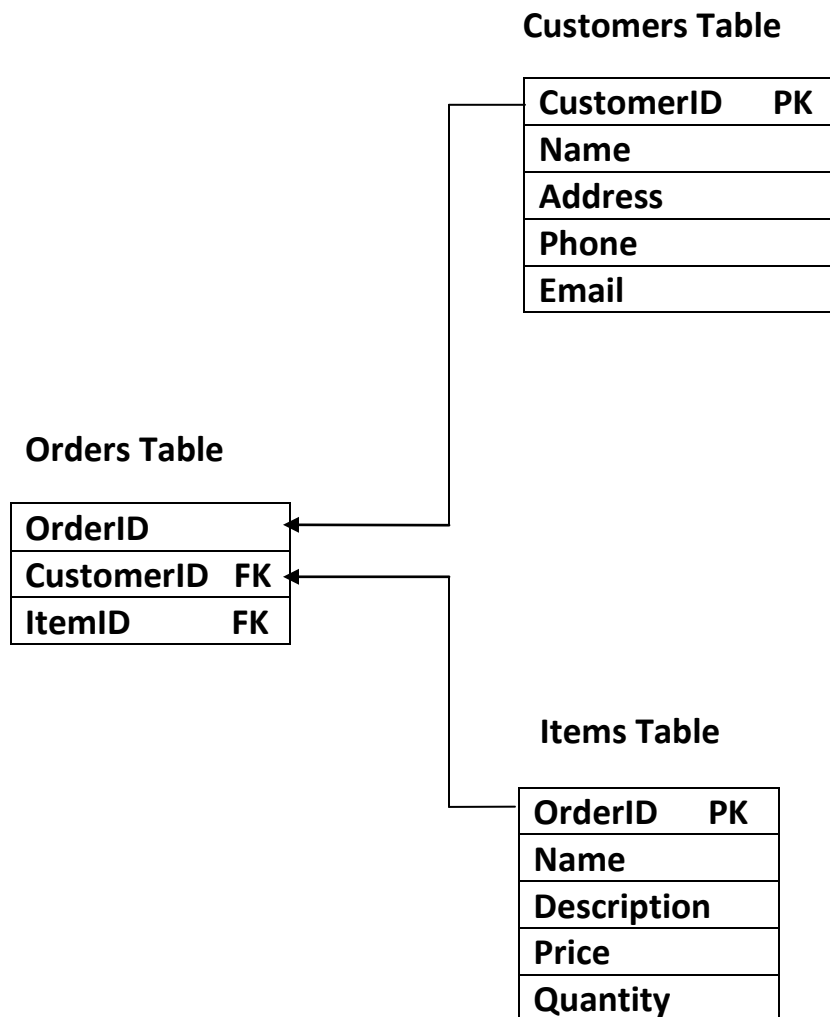
The ItemID of the Items table is a Primary Key

The CustomerID of the Customers table is a Primary Key

The ItemID of the Items table is a Foreign Key

The CustomerID of the Items table is a Foreign Key

Here is the table relationships:



The primary key of the Customers Table is the CustomerID

The primary key of the Items Table is OrderId

The orders table has the foreign keys CustomerID and OrderID.

The customerID represents the customer information for the order, and the OrderID represents the Item ordered. An order may have many different customers and order items.

The orders table does not have a primary key because customer and Orders ID are not unique.

We first make the Create the SQL statement for the Customers Table

```
sql_customers_table = """"
CREATE TABLE IF NOT EXISTS Customers (
    CustomerId INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    Name TEXT NOT NULL,
    Address INTEGER NOT NULL,
    Email NOT NULL,
    PHONE NOT NULL
);
""""
```

We use the triple """" so we can enclose multi-line strings

We have made the CustomerID as the primary key and to be **AUTOINCREMENT NOT NULL** means the value must be inserted.

Next we make the SQL Create statement for the Items Table

```
sql_items_table = """"
CREATE TABLE IF NOT EXISTS Items (
    ItemId INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    Name Text NOT NULL,
    Description Text NOT NULL,
    Price REAL NOT NULL,
    Quantity INTEGER NOT NULL
);
""""
```

We have made the ItemID as the primary key and to be **AUTOINCREMENT**

Lastly we make the SQL Create statement for the Orders Table

```
sql_orders_table = """  
CREATE TABLE IF NOT EXISTS Orders  
(  
    OrderID INTEGER NOT NULL,  
    CustomerId INTEGER NOT NULL,  
    ItemId INTEGER NOT NULL,  
    Quantity INTEGER NOT NULL,  
    Total REAL NOT NULL,  
    FOREIGN KEY(CustomerId) REFERENCES Customers(CustomerId),  
    FOREIGN KEY(ItemId) REFERENCES Items(ItemId)  
);  
"""
```

The orders table does not have a primary key but just an OrderId. We do not use a primary key because there will be many order records all having the same OrderId.

The Orders table has 2 foreign keys:

```
CustomerId INTEGER NOT NULL,  
ItemId INTEGER NOT NULL,
```

The reference to the primary key is listed at the bottom of the Order Table Create Statement

```
FOREIGN KEY(CustomerId) REFERENCES Customers(CustomerId),  
FOREIGN KEY(ItemId) REFERENCES Items(ItemId)
```

Next we create the tables from the SQL Create statements

```
# execute sql create statement's  
cursor.execute(sql_customers_table)  
cursor.execute(sql_items_table)  
cursor.execute(sql_orders_table)
```

```
# commit database to others  
conn.commit()
```


Once we make the Tables we need to insert the values into the tables

Inserting values into the Items table:

```
# insert data int items table
sql = "INSERT INTO Items
( Name, Description, Price, Quantity) VALUES ('Apples', 'Shinny Red',1.95,100)"
cursor.execute(sql)
itemid1 = cursor.lastrowid

sql = "INSERT INTO Items
( Name, Description, Price, Quantity) VALUES ('Oranges', 'Sweet and Ripe',2.95,100)"
cursor.execute(sql)
itemid2 = cursor.lastrowid
```

Notice we have inserted the items without using the Item id, because the ItemID is automatically generated for you.

We can obtain the auto generated ItemID from by using **lastrowid** property from the cursor object.

```
itemid1 = cursor.lastrowid
itemid2 = cursor.lastrowid
```

to do:

Print items tables

Inserting values into the Customers table:

To insert values into the customer table we use the INSERT SQL command.

```
# insert data into customers
sql = "INSERT INTO Customers ( Name, Address, Email,Phone) VALUES
('Sue Jones', '1 One St','645-8654','sue@mail.com')"
cursor.execute(sql)
customerid1 = cursor.lastrowid

sql = "INSERT INTO Customers ( Name, Address, Email,Phone) VALUES
('Mary Berry', '7 Hanover St','454-5432','mary@mail.com')"
cursor.execute(sql)
customerid2 = cursor.lastrowid
```

Notice we have inserted the items without using the CustomerID because the CustomerID is automatically generated for you.

We can obtain the auto generated CustomerID from the **lastrowid** property from the cursor object.

```
customerid1 = cursor.lastrowid
customerid2 = cursor.lastrowid
```

to do:

print out the CustomersTable

Inserting values into Orders Table

To insert data into the Orders Table we must know the primary keys of the Customerid and Item id which we previously obtained using

```
cursor.lastrowid
```

We also can get the prices from the items table using the itemid1 and itemid2 and the **fetchone** method. We use the **fetchone** method since we only need 1 value.

```
# get prices from items
sql = "select price from items where ItemId = ?"
price1 = cursor.execute(sql,(itemid1,)).fetchone()[0]
sql = "select price from items where ItemId = ?"
price2 = cursor.execute(sql,(itemid2,)).fetchone()[0]
```

Once we obtain primary keys id's and the item prices we can insert them into the Orders table.

```
# insert data into orders
sql = "INSERT INTO orders (OrderId,CustomerId,ItemId,Quantity,Total) VALUES (?, ?, ?, ?, ?)"
cursor.execute(sql,(1,customerid1,itemid1,2,price1*2))
sql = "INSERT INTO orders (OrderId,CustomerId,ItemId,Quantity,Total) VALUES (?, ?, ?, ?, ?)"
cursor.execute(sql,(1,customerid1,itemid2,3,price2*3))
sql = "INSERT INTO orders (OrderId,CustomerId,ItemId,Quantity,Total) VALUES (?, ?, ?, ?, ?)"
cursor.execute(sql,(1,customerid2,itemid1,4,price1*4))
sql = "INSERT INTO orders (OrderId,CustomerId,ItemId,Quantity,Total) VALUES (?, ?, ?, ?, ?)"
cursor.execute(sql, (1,customerid2,itemid2,2,price2*2))
```

to do:

Print out OrdersTable

Run your program, you should get something like this:

Customers

(1, 'Sue Jones', '1 One St', '645-8654', 'sue@mail.com')

(2, 'Mary Berry', '7 Hanover St', '454-5432', 'mary@mail.com')

Items

(1, 'Apples', 'Shinny Red', 1.95, 100)

(2, 'Oranges', 'Sweet and Ripe', 2.95, 100)

Orders

(1, 1, 2, 1.95)

(1, 2, 3, 2.95)

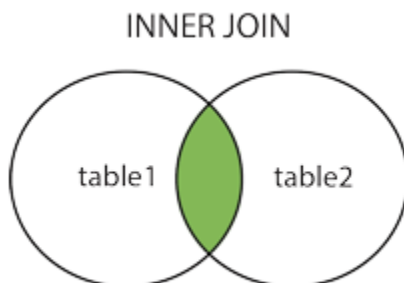
(2, 1, 4, 1.95)

(2, 2, 2, 2.95)

You can see that the orders table contain the order key of the customers and item table as foreign keys.

INNER JOIN

Inner join allows you to join tables together usually using a **select** statement.



The syntax of inner join of 2 tables:

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

The column names can be from either table. The join on, usually connects primary keys to foreign keys. The tables can be listed in any order.

Example using inner join of 2 tables

Here we are joining the Customers table to the Orders table so we can print out the customer name and order quantity and total price.

```
# inner join 2 tables
# select customer names and orders
sql = """SELECT Customers.Name,Orders.OrderID,Orders.Quantity,Orders.Total
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID
"""
```

We then print out the results:

```
rows = cursor.execute(sql).fetchall()

# print out rows
for row in rows:
    print(row)
```

```
('Sue Jones', 1, 2, 3.9)
('Sue Jones', 1, 3, 8.8500000000000001)
('Mary Berry', 1, 4, 7.8)
('Mary Berry', 1, 2, 5.9)
```

Inner join using a where clause

With a where clause you can just select specific records.

```
# select customer names and orders
sql = """SELECT Customers.Name,Orders.OrderID,Orders.Quantity,Orders.Total
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID where
Customers.Name = 'Sue Jones'
"""
```

We then print out the results:

```
rows = cursor.execute(sql).fetchall()

# print out rows
for row in rows:
    print(row)
```

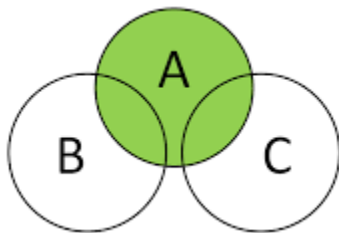
```
('Sue Jones', 1, 2, 3.9)
('Sue Jones', 1, 3, 8.8500000000000001)
```

TO DO:

Join the Orders Item table with the Orders Table . You should print out the item name and description. Then use a where clause to print out the Order Item for a specified item id.

Inner join on 3 tables

Inner joining of 3 tables makes it much easier to print out information for all 3 tables all at once.



The syntax of inner join of 3 tables:

```
SELECT column_name(s)
FROM ((table1
INNER JOIN table2
ON table1.column_name = table2.column_name)
INNER JOIN table3
ON table1.column_name = table2.column_name);
```

The column names can be from either table. The join **on**, usually connects primary keys to foreign keys. The tables can be in any order.

Using **inner join** we can join the Customer table to the Items table so we can print the customer and order item for each order in the order table.

```
sql = """SELECT Orders.OrderID,Customers.Name, Items.Name,Orders.Quantity,Orders.Total
FROM ((Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID)
INNER JOIN Items ON Orders.ItemID = Items.ItemID);
"""
```

We then print out the results:

```
rows = cursor.execute(sql).fetchall()
```

```
# print out rows
for row in rows:
    print(row)
```

```
(1, 'Sue Jones', 'Apples', 2, 3.9)
(1, 'Sue Jones', 'Oranges', 3, 8.85)
(1, 'Mary Berry', 'Apples', 4, 7.8)
(1, 'Mary Berry', 'Oranges', 2, 5.9)
```

To do

Use a where clause to select Customer name and/or item name.

CONSTRAINTS

Constraints prevent the data base from changing if the following transactions involving primary keys and foreign keys are violated.

- (1) You cannot enter a row into a table if the foreign keys does not exist.
- (2) You cannot delete a row from a table if those foreign keys are used in another table

For constraints to work in sqlite you need to activate them right after you connect to the database.

```
conn.execute("PRAGMA foreign_keys = 1")
```

like this:

```
# connect to data base  
conn = sqlite3.connect("lesson11.db")  
# activate constraints  
conn.execute("PRAGMA foreign_keys = 1")
```

Here is our test program to test constraints:

In this situation we are deleting itemID primary key 1 from the item table that is used in the Orders Table as a foreign key.

```
# test constraints  
  
# delete a row where the primary key is used in another table as a foreign key  
itemid = 1  
sql = "DELETE FROM Items WHERE ItemID = ?"  
cursor.execute(sql,(itemid,))
```

```
cursor.execute(sql,(itemid,))  
sqlite3.IntegrityError: FOREIGN KEY constraint failed
```

In this next situation we are inserting an unknown itemID foreign key 5 for the Item table that is used in the Orders table.

```
# insert data into orders with an unknown itemID  
  
sql = "INSERT INTO orders  
(OrderId,CustomerId,ItemId,Quantity,Total) VALUES (?,?,,?,?)"  
  
cursor.execute(sql,(1,8,5,2,price1*2))
```

```
cursor.execute(sql,(1,8,5,2,price1*2))  
sqlite3.IntegrityError: FOREIGN KEY constraint failed
```

Lesson 11 Homework Part2

Create a Database that has a Videos table

A **Videos** table will have a VideoID, Title, Price and Quantity in stock.

The VideoID should be a INTEGER Primary Key Auto Increment

Title is TEXT, Price is REAL and Quantity is INTEGER. Make all fields NOT NULL.

Make a **Clients** table that has a ClientID, name TEXT and an INTEGER representing number of books that have been rented out. The ClientID should also be a INTEGER Primary Key Auto Increment.

Make a table called **Rentals** that has a RentalID and two columns to store the primary keys of the Video table and Client table as INTEGER NOT NULL foreign keys. Also make a column to store true (1) or false (0) where true means the video is rented out, and false meaning the video has been returned.

Optionally make classes Video, Client and Rental to move the data in and out of the data base.

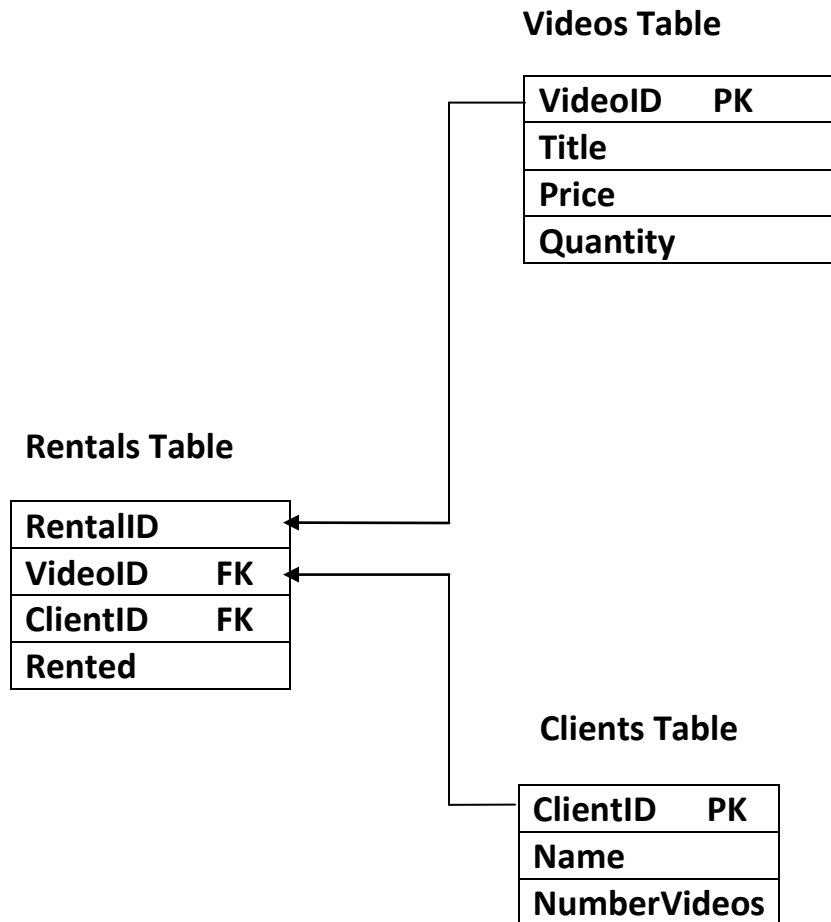
Make a main program that will connect to the database, create the tables and insert some videos to rent and allow clients to rent out videos and return them.

Make a menu the displays videos to be rented out, rent a video, return a video and display videos that have been rented out and list clients. Do not display video to be rented out if their quantity is zero. Do not rent videos that have quantity 0.

When a video is rented out decrement the quantity in the video table and increment the number of books rented out in the client table. Store the rental status in the rental table as rented out. When the book is returned set the status in the rental table as returned, decrement the quantity on the client table and increment the quantity in the video table.

Call your py file lesson11b.py or videostore.py and your data base lesson11b.db or videostore.db. Make sure you use constraints.

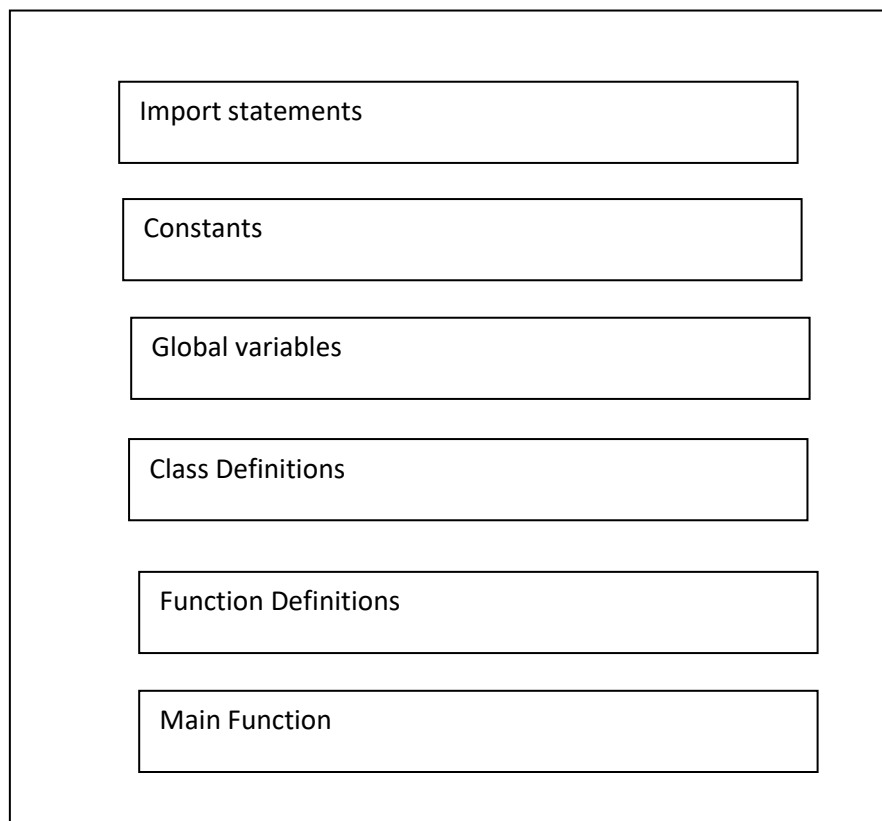
Your videostore data base diagram may look like this:



PYTHON PROJECTS

This is where all the things you learned previously connect together. Now everything will make sense and understanding will be enforced. You previously learned what all the programming statements do, and now it is the time you use them. Recapping: when you want to store some values you use a **variable**. When you want to print a value to the screen you use a **print** statement. When you want to get a value from the key board you use a **input** statement. When you want to store a sequence of values (like a shopping list) you use a **list**. When you want to store a sequence of unique items you want to use a **set**. When you want to store a value associated with another value (like a phone book) you use a **dictionary**. When you want to group a bunch of programming statements together to do a certain task you use a **function**. To return more than one value from a function you use a **tuple**. To store common variables together and do operations on them you use a **class**. Python program structure to follow for projects:

Python Program Layout



Project 1 Spelling Corrector

Read in a text file with spelling mistakes, find the incorrect spelled words and offer corrections. The user should be able to choose the correct choice from a menu. Look for missing or extra letters or adjacent letters on the keyboard. Download a word dictionary from the internet as to check for correct spelled words. Use a dictionary to store the words. Store the correct spelled file.

Project 2 Math Bee

Make a Math bee for intermixed addition, subtraction, multiplication and division single digit questions. Use random numbers 1 to 9 and use division numbers that will divide even results. Have 10 questions and randomly generate numbers and operations. . Keep track of the users score. You will need to use the python random class to generate random numbers.

You first need to import using:

```
import random
```

Then you call **randint** function from the random class, giving a start value and a end value.

```
x = random.randint(1,10)
```

Project 3 Quiz App

Make a quiz app with intermixed multiple choice, true and false questions.

You should have a abstract Question super class and two derived classes MultipleChoice and TrueAndFalse. Each derived class must use the abstract methods to do the correct operation. An abstract method is a method that has no code and just contains a **pass** statement. An abstract class only contains abstract methods. An abstract class cannot be instantiated. Store all questions in one file. Store the results in another file indicating the quiz results.

Project 4 Phone Book App

Make a phone book app that uses a dictionary to store names and phone numbers. Use the name as the dictionary key and the phone number as the dictionary value. You should be able to view, add, delete contacts as menu operations. Contact names need to be displayed in alphabetical order by name. Offer to lookup contacts by name or by phone number. Contacts should be stored in a CSV file or SQL Contact Table, read in when app runs, and saved when the app finished running. Use a separate function to make a selection menu that returns a user selection. Bonus: View contacts by name or phone number. You may want to make a optional Phone Book class.

Project 5 Address Book App

Make a address book app that uses a dictionary to store names and contact information. You need a Contact class to store name and address, email and/or phone number. Store the Contacts in a Dictionary. Use the name as the dictionary key and the Contact object as the dictionary value. You should be able to view, add, delete contacts as menu operations.

Make a menu in a separate function that returns the user selection.

1. List contacts by name
2. List contacts by address, email or phone
3. Add contact
4. Search for Contact by name
5. Search for Contact by address, email or phone
6. Remove Contact by name
7. Remove Contact by address, email or phone
8. Exit

Contacts need to be displayed in alphabetical order by name or address. Contacts should be stored in a CSV file or SQL table , read when app runs, and saved with app finished running.

You can use the sort or sorted list function and a lambda function to sort the list of contacts as follows:

```
# sort the list in place ascending order by email  
contacts.sort(key=lambda x: x.email)
```

```
# return a new list ascending order by email  
contacts = sorted(contacts, key=lambda x: x.email)
```

You can use the **reverse=True** argument to sort the list in descending order

```
# sort the list in place descending order by email  
contacts.sort(key=lambda x: x.email, reverse=True)
```

```
# return a new contact list, using the sorted() list function  
# descending order by email  
contacts = sorted(contacts, key=lambda x: x.email, reverse=True)
```

. You may want to make a optional Address Book class.

Project 6 Appointment App

Make an Appointment book app that uses a dictionary to store Appointments. You need an Appointment class to store name, description, date and time. You should be able to view, add, delete, scroll up and down appointments as menu operations. Appointments need to be displayed in chronological orders. Appointments should be stored in a CSV file or SQL table, read when app runs, and saved with app finished running. Save the appointments as csv files. Use a separate function to make a selection menu that returns a user selection. You will also need to use the python datetime class. The datetime class store both date and time.

You first import the datetime class using

```
import datetime
```

You can get the date and time for today with

```
dt = datetime.datetime.now()
```

You can print out the date object like this

```
print(dt) # 2018-07-31 13:41:33.332930
```

Or format it like this using the **strftime** function like this:

```
print(dt.strftime("%a %b %d %H:%M:%S %Y")) # Tue Jul 31 13:41:33 2018
```

You can make your own date time object from known year, month, day, hours, minutes and seconds like this

```
d = datetime.datetime(year, month, day, hour, minute, second)
```

You can extract the date and times from the datetime object using the datetime object getter functions

```
year = dt.year
```

```
month = dt.month
```

```
day = dt.day
```

```
hour = dt.hour
```

```
minute = dt.minute
```

```
second = dt.second
```

You can convert a string to a date time object using the **strptime** function like this using the input format : "%Y-%m-%d %H:%M:%S"

```
dt = datetime.datetime.strptime("2018-07-31 13:41:33", "%Y-%m-%d %H:%M:%S")
```

```
print(dt) # 2018-07-31 13:41:33
```

You can compare date objects using the standard compare operators < and >

```
print (dt > dt2) # False
```

```
print (dt < dt2) # True
```

You can subtract 2 dates like this using the subtract - operator and returns days and time

```
print (dt - dt2) # -1 day, 23:33:50.608299
```

You may want to make a optional AppointmentApp class.

Project 12 Grocery Store App

Make a Grocery Store App where Customers can purchase items. Preferred customers get a discount. After all items have been entered a receipt is printed.

Step 1: Item class

Make a Item class with private variables product name, quantity ordered, price and discount price.

If the item is not a discount item then the discount price is 0.

Make a constructor that will receive the item name, price and discount price.

Make getters and setters for each instance variable

Make a formatted `__str__()` method that will return item name price quantity discount price surrounded by round brackets and extension price like this:

```
Carrots      2    1.29 (.89)  2.58 (1.78)
```

Step 2: GroceryStore class

Make a GroceryStore class that will store items bought, total items bought, total the order and print out a receipt.

The Grocery Store class will store the customer's name, and all items bought in an array list.

The Grocery store constructor will receive the customer name.

The grocery store will have a method to add an item object.

The Grocery store class will have a Get Total method, to be used to print out the receipt

The grocery store class will also print out a receipt using the printReceipt method.

All instance variables are private and cannot have any getters and setters.

Step 3: DiscountStore class

The DiscountStore class inherits from the GroceryStore class

If the customer is a preferred customer then the DiscountStore class is used.

The DiscountStore class has methods to calculate discount percent, and count of discount items. The discount class will override the getTotal class of the grocery store class.

The discount store class will also print a receipt showing the number of discount, items and the discount percent obtained.

Step 4 GroceryApp class.

The GroceryApp class is the main class where the cashier enters the customer items, bought.

The cashier will ask if the customer is a preferred customer. if it is a preferred customer then the DiscountStore class is used else us the Grocery Store class Is used.

The cashier will enter the items bought. Once all items have been enters then the receipt is printed out.

You will need to store a list of products in a file or SQL table to simulate the entering of products.

The file will be like this:

Customer name

Preferred or not preferred

Number of products

Item name, quantity, price, discount price

Example file:

Tom Smith

Preferred

3

Carrots , 2.49, 1.78, 2

Fish,12.67, 11.89,3

Milk 4.89.3.75, 2

The main method will have a menu as follows:

- (1) read order file
- (2) print receipt
- (3) store receipt to a file
- (4) exit program

END