Python Mini Lessons

last update: April 23, 2020

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.

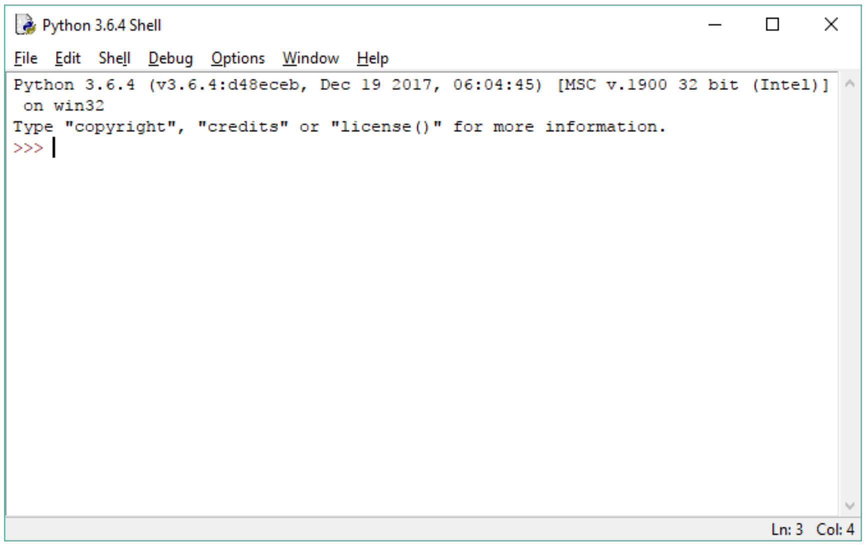
Let's get started!

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.

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

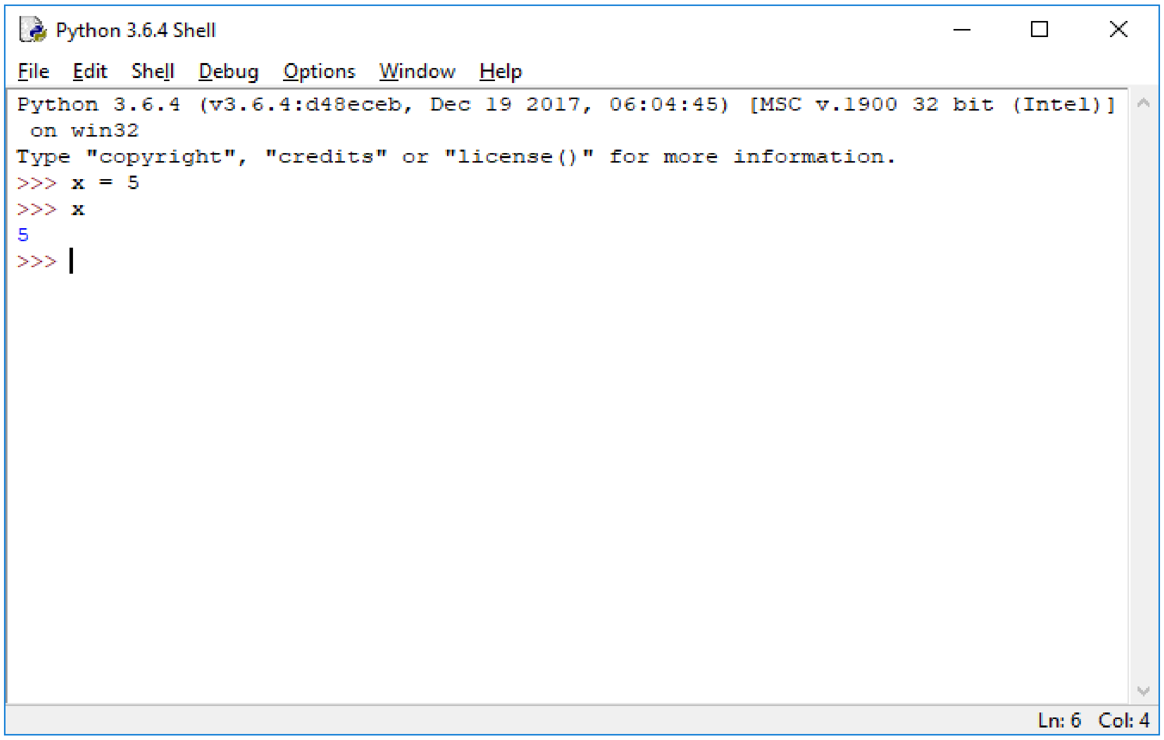


```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
 on win32
Type "copyright", "credits" or "license()" for more information.
>>> 
```

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:

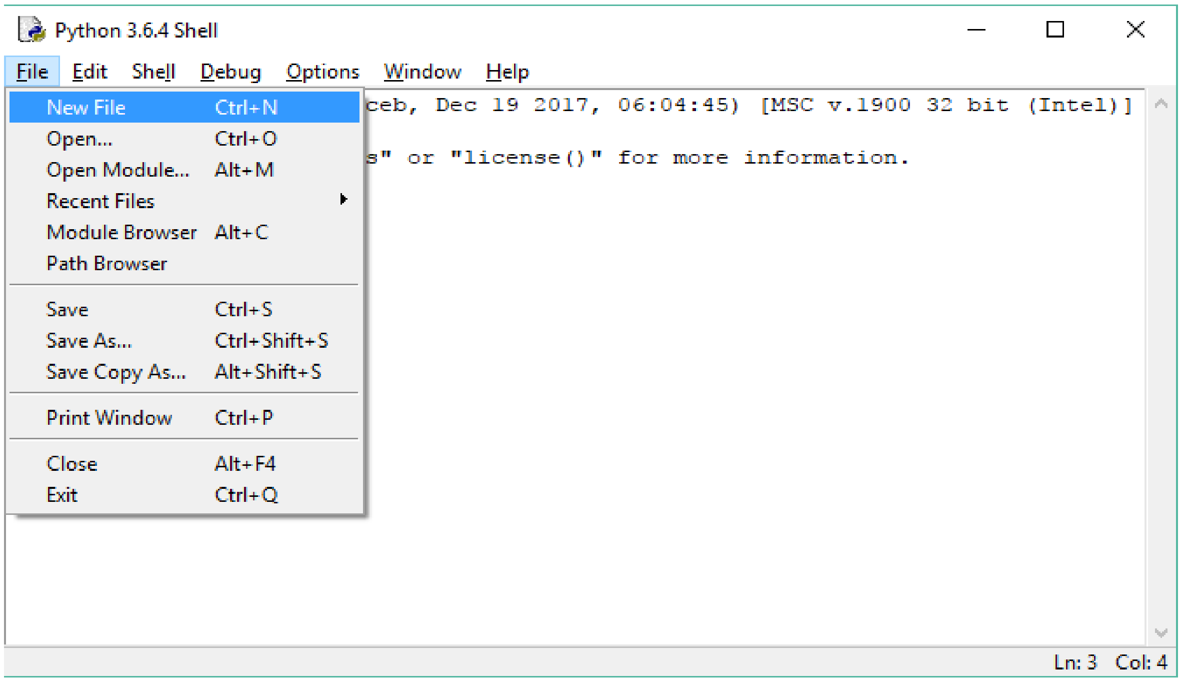


```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
 on win32
Type "copyright", "credits" or "license()" for more information.
>>> x = 5
>>> x
5
>>> 
```

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.

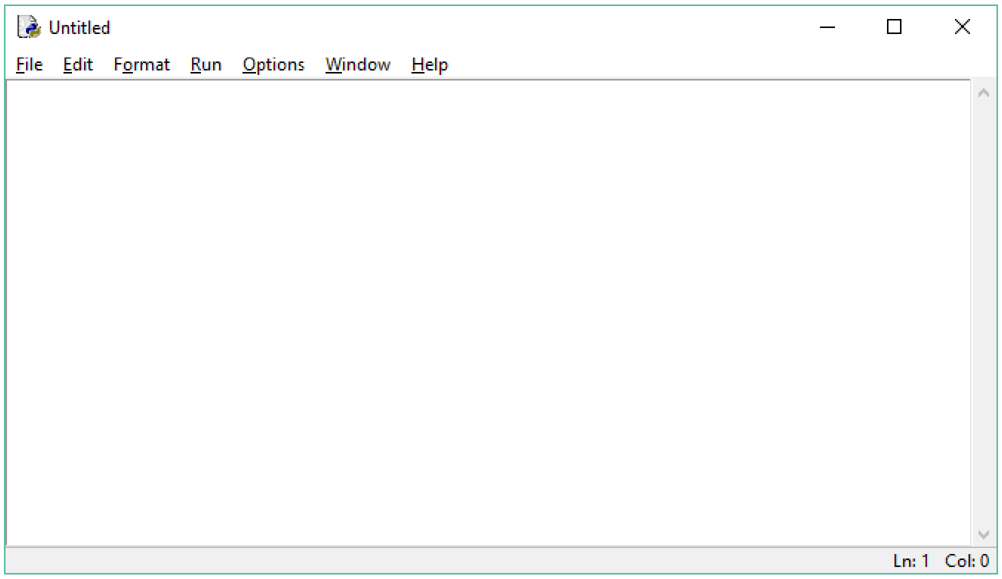
Programming statements

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.

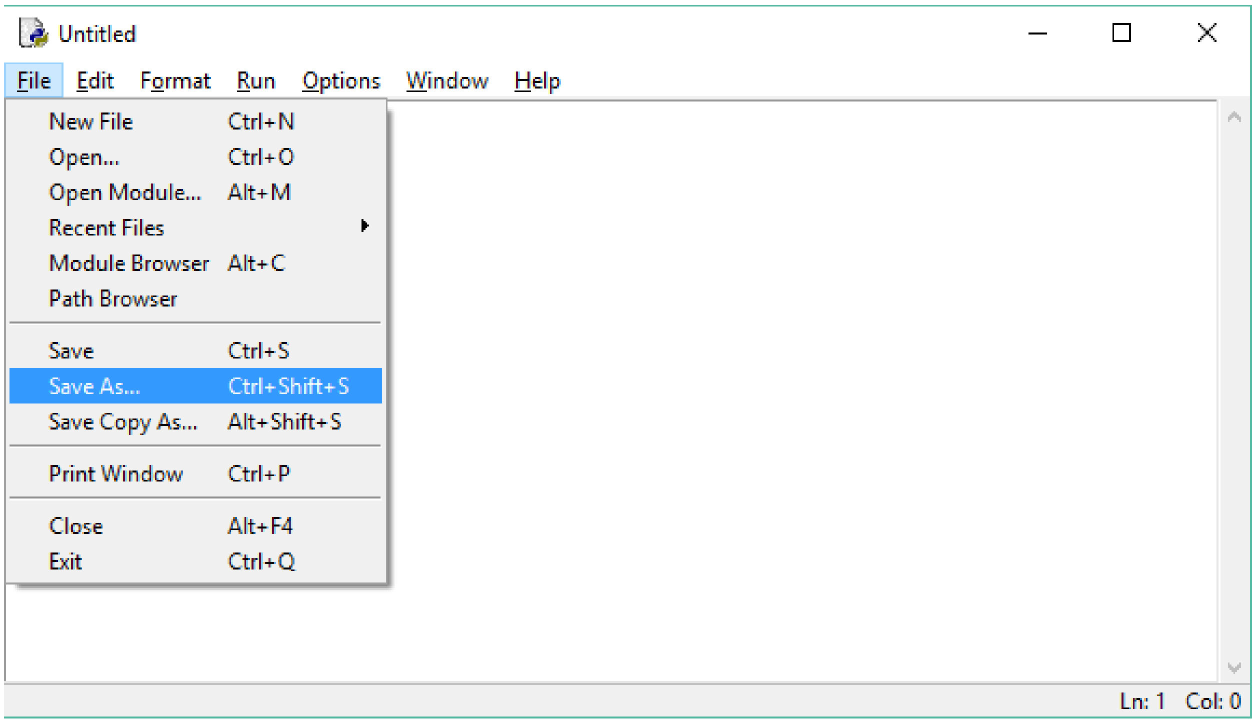


```
ceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
s" or "license()" for more information.
```

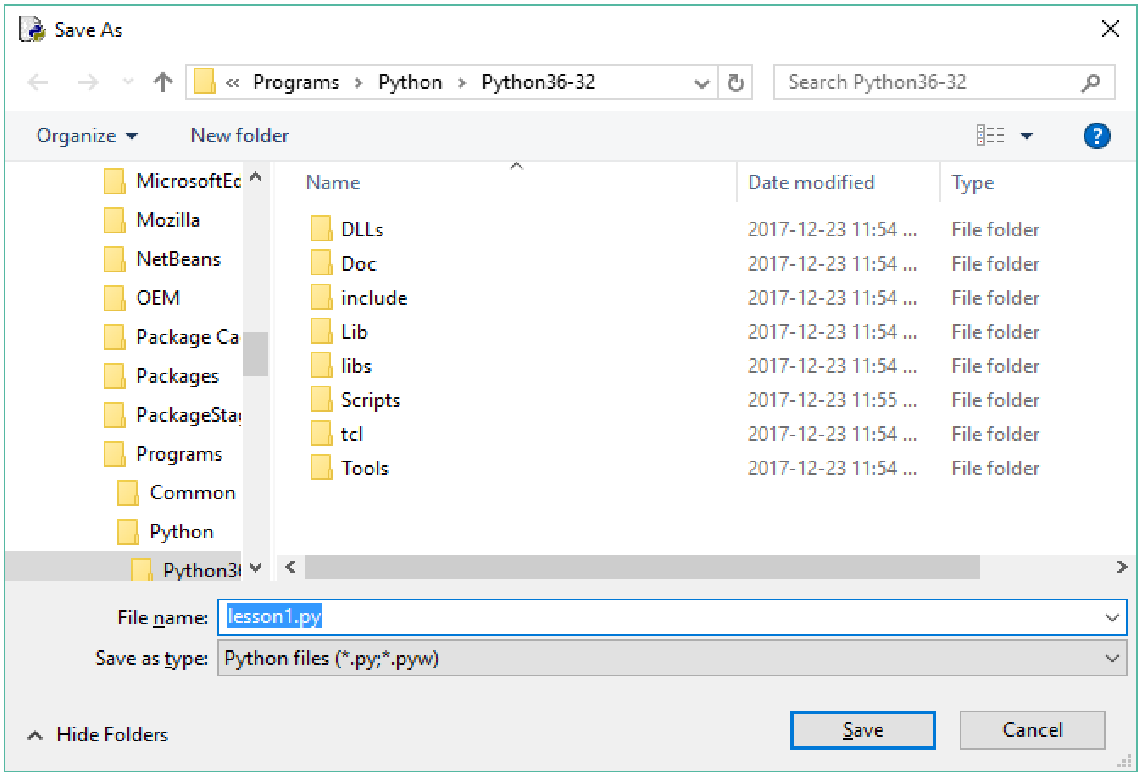
The editor window appears where you can type in Python programming statements that you can save and run. You may want first to make a folder on your computer called python Lessons to store all your python programs.



From the File menu select File Save As



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

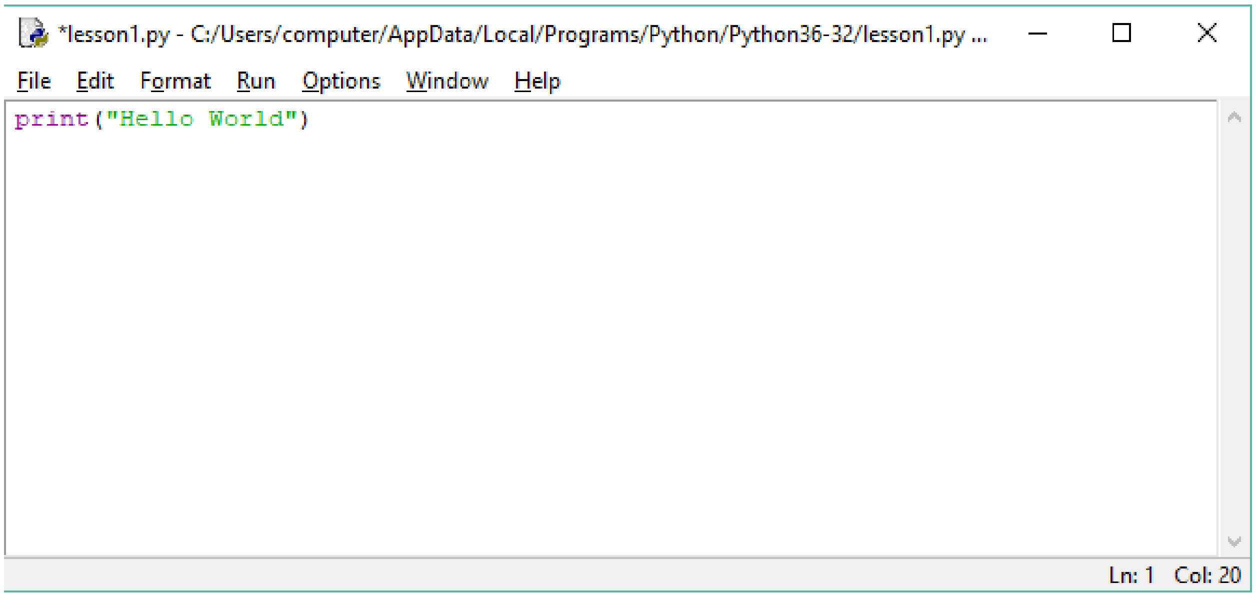


Name	Date modified	Type
DLLs	2017-12-23 11:54 ...	File folder
Doc	2017-12-23 11:54 ...	File folder
include	2017-12-23 11:54 ...	File folder
Lib	2017-12-23 11:54 ...	File folder
libs	2017-12-23 11:54 ...	File folder
Scripts	2017-12-23 11:55 ...	File folder
tcl	2017-12-23 11:54 ...	File folder
Tools	2017-12-23 11:54 ...	File folder

File name: lesson1.py
Save as type: Python files (*.py;*.pyw)

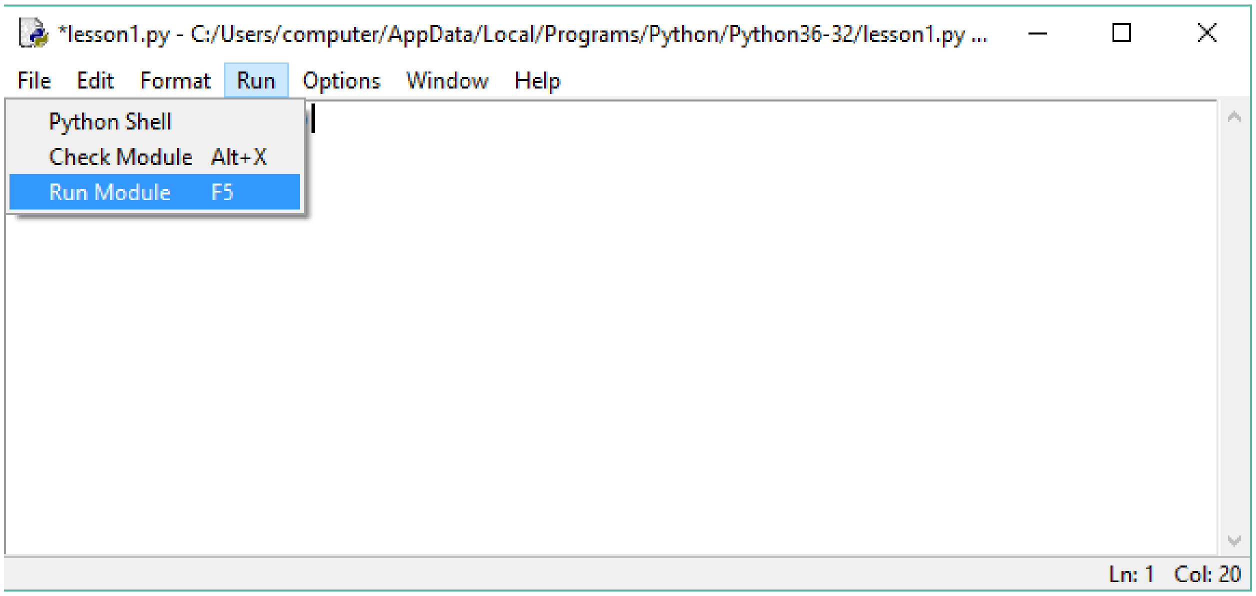
It’s now time to write your first Python Programming Statement. Your program will print Hello World on the screen. In the Python Editor type:

print("Hello World")

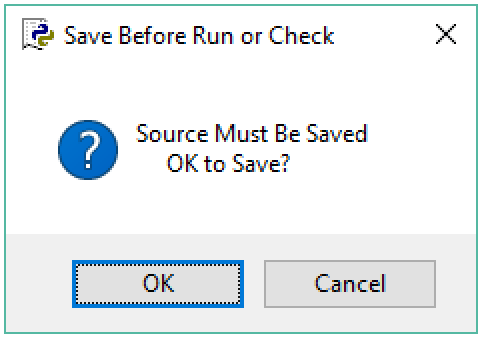

*lesson1.py - C:/Users/computer/AppData/Local/Programs/Python/Python36-32/lesson1.py ...
File
Edit
Format
Run
Options
Window
Help
Ln: 1
Col: 20

```
print("Hello World")
```

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

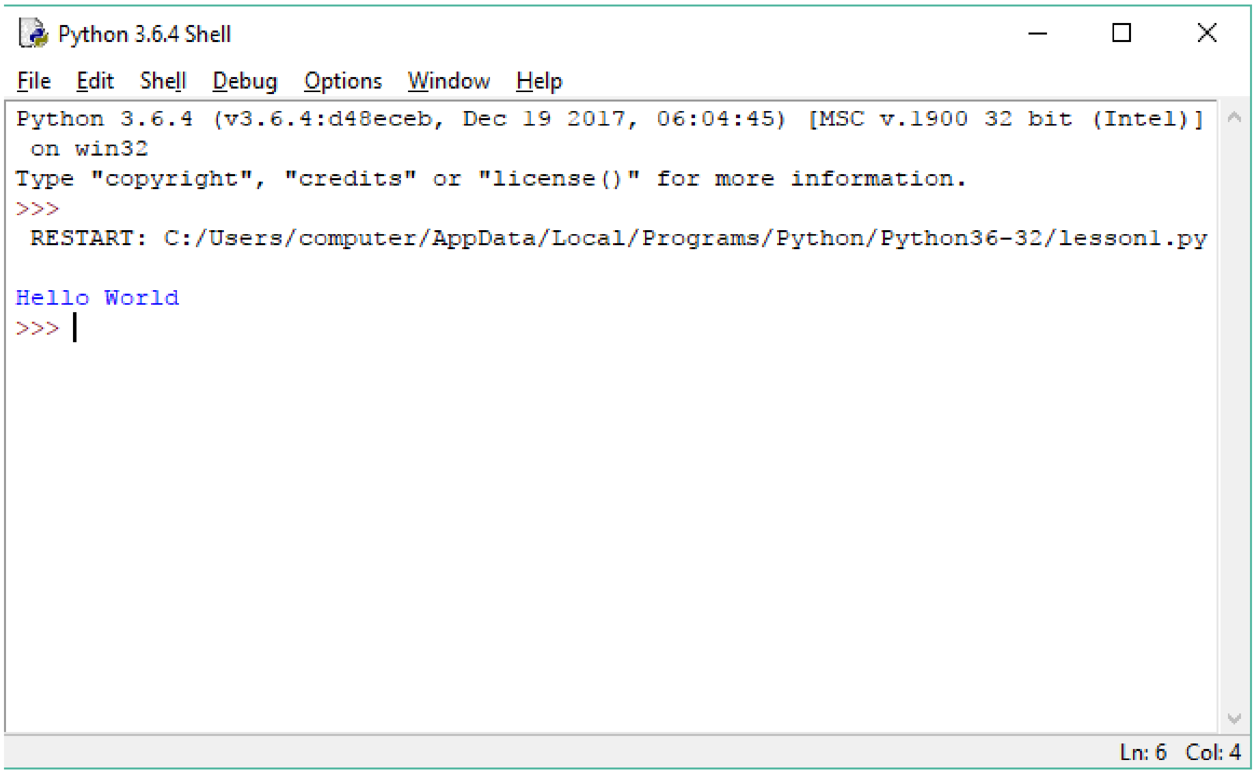

*lesson1.py - C:/Users/computer/AppData/Local/Programs/Python/Python36-32/lesson1.py ...
File Edit Format Run Options Window Help
Python Shell
Check Module Alt+X
Run Module F5
Ln: 1 Col: 20

Select OK



Source Must Be Saved
OK to Save?

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



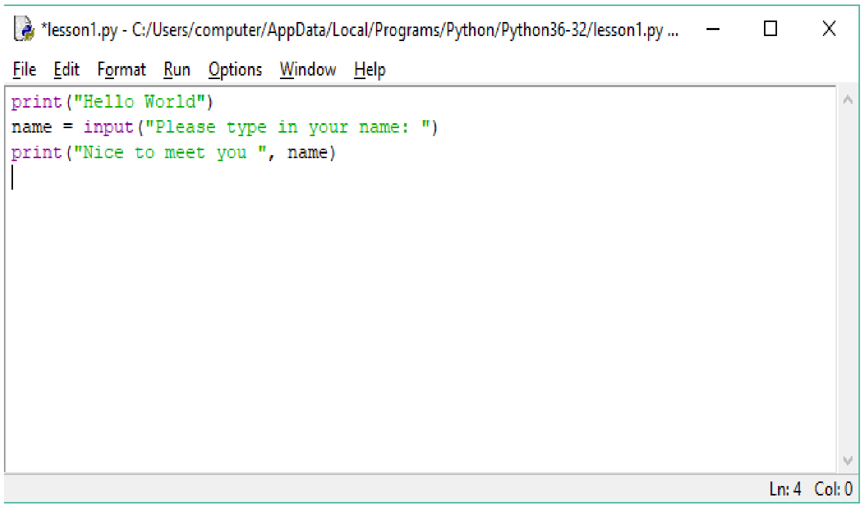
```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
 on win32
Type "copyright", "credits" or "license()" for more information.
>>> 
 RESTART: C:/Users/computer/AppData/Local/Programs/Python/Python36-32/lesson1.py

Hello World
>>> 
```

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: ")

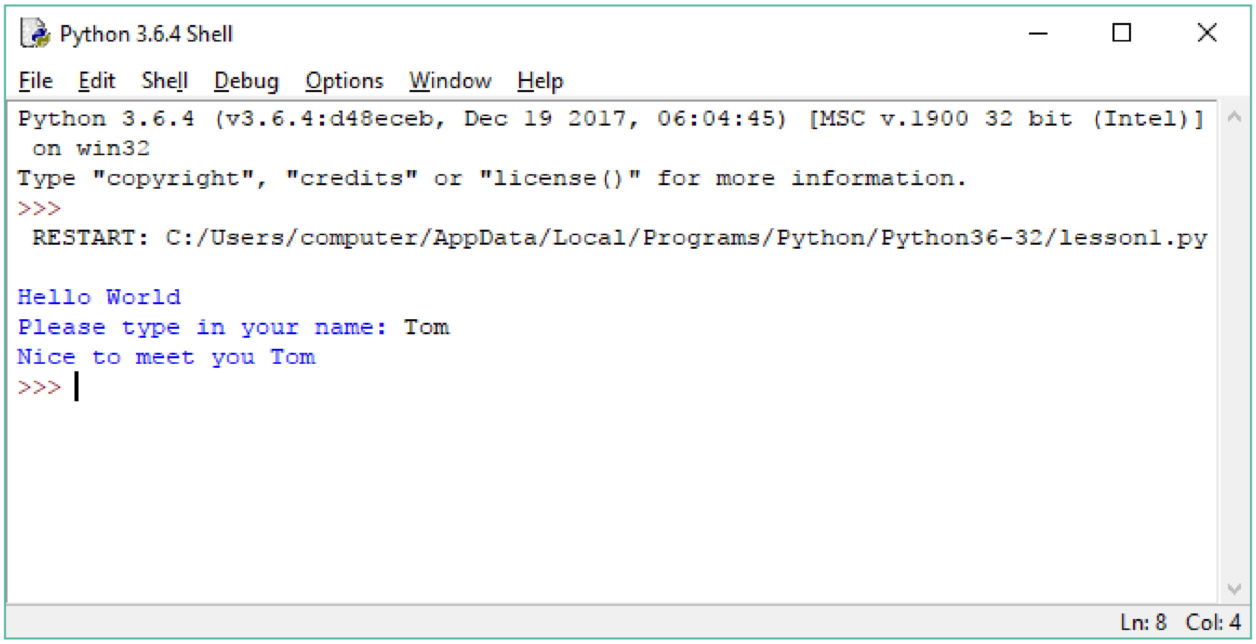
print("Nice to meet you ", name)


*lesson1.py - C:/Users/computer/AppData/Local/Programs/Python/Python36-32/lesson1.py ...
File Edit Format Run Options Window Help
Ln: 4 Col: 0

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

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

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



```
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
>>> 
```

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 persons 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)

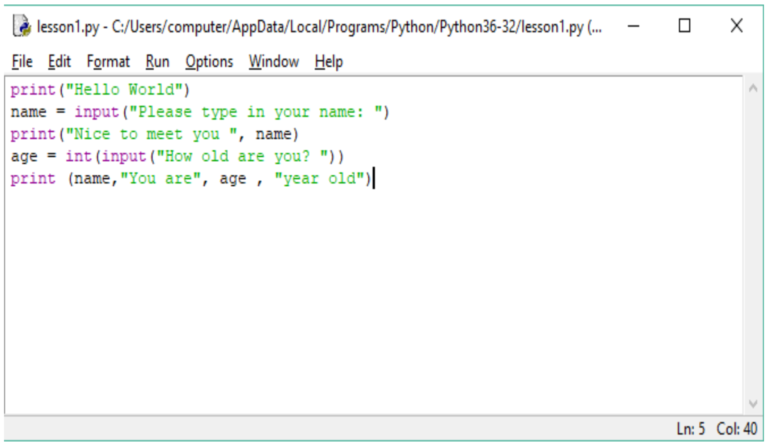
Note inside the print statement the string message and variable name are enclosed in round brackets and the values are separated by commas. Round brackets are important in Python. The opening round bracket ‘(‘ introduces the start of the values, the closing ‘)’ round bracket species the end of the values.

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 (name,"You are", age , "year old")

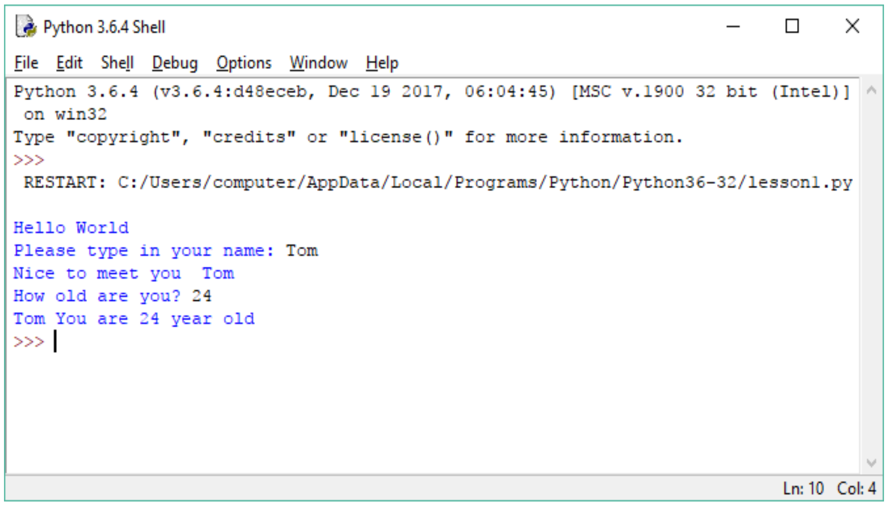


lesson1.py - C:/Users/computer/AppData/Local/Programs/Python/Python36-32/lesson1.py (...

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

Ln: 5 Col: 40

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



Python 3.6.4 Shell

```
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

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 , "year 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 i**nt** statement or **float** statement on the **input** statement to convert the input value to a numeric value.

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("Hello World")
        name = input("Please type in your name: ")
        print("Nice to meet you ", name)
        age = int(input("How old are you? "))
        print (name,"You are", age , "year 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 them 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.

Homework:

Write a python program that ask someone what their profession is like Doctor, Lawyer, Salesman etc. and how much money they make, then print out their details. Use a main function to call your programming statements. Call your Python homework program, homework1.py

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.

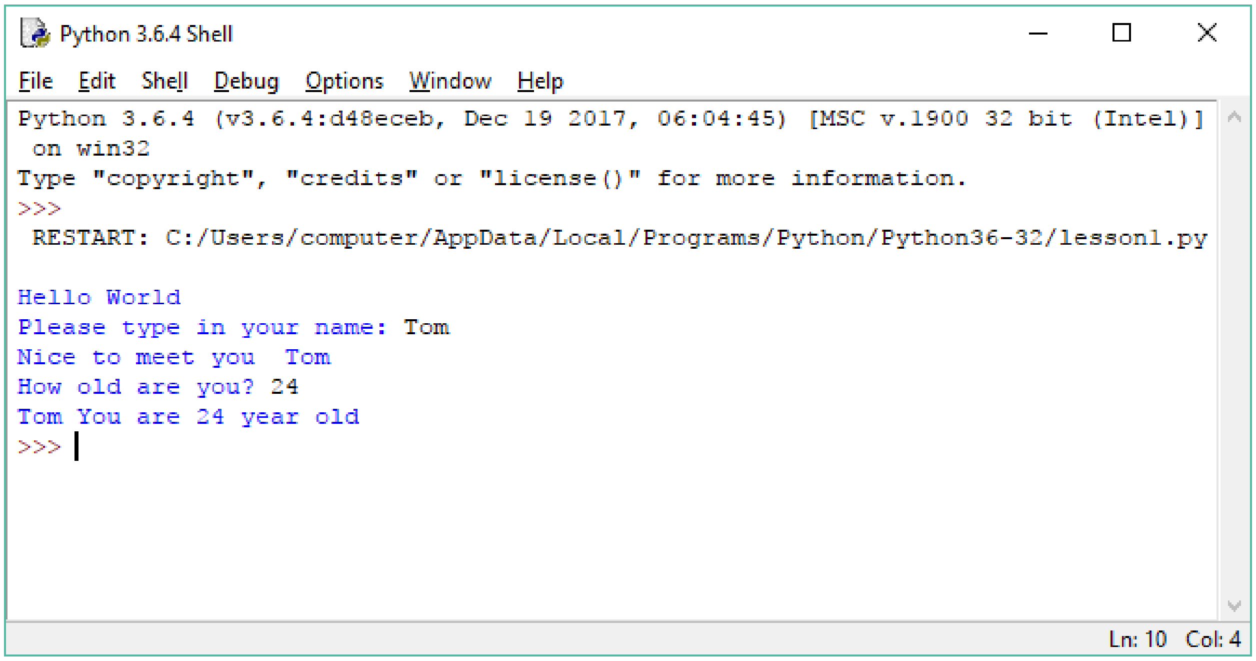
Functions usually are defined at the top of the program in order as they are used. The main function is the last one because it will call all the proceeding functions. When a function is called in a programming statement it means it is executed. In a Python script the functions must be defined before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

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

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

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


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


def printPerson(name, age):
    print("Nice to meet you ", name)
    print (name,"You are", age , "year old")
```

```
def main():

    welcome()

    name = enterName()

    age = enterAge()

    printPerson(name, age)

main()
```

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



```
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
>>> 
```

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 printDetails function receives a name and age value to print out, but return's no value.

```
def  printPerson(name, age):
    print("Nice to meet you ", name)
    print (name,"You are", age , "year old")
```

The **name** and **age** inside the round brackets of the **printDetails** 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 printDetails function.

```
def main():

    welcome()
    name = enterName()
    age = enterAge()
    printPerson(name, age)
```

Notice, our main function is must 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 are 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("Hello World")


# 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 printPerson(name, age):

    print("Nice to meet you ", name)

    print (name,"You are", age , "year old"
# main function to run program
def main():
   welcome()  # welcome user
   name = enterName() # get user name
   age = enterAge() # get user age
   printPerson(name, age) # print user name an age


main() # call main function to run program
```

Homework:

Take your homework program from Lesson1.py and use functions. Make functions welcome, enterProfession, enterSalary, printEmployee and main. You can call your python homework program homework2.py

<u>Lesson 3</u>   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 on these variables that store the values. 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):

        sout = "Nice to meet you " + self.name + "\n"
        sout += self.name + " You are " +  str(self.age) +  " years old"

        return sout
```

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 o the Person class and are not the same ones in the Person class.

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):
    sout = "Nice to meet you " + self.name + "\n";
    sout += self.name + " You are " +  str(self.age) +  " years old"
    return sout
```

Notice we have no print statement in our **__str__** function. We assign information to the local variable sout and return the sout value. A local variable is just known to the function it resides in. The sout 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 function 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.

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.

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

We can also print our the persons name and age by calling the __str__ function automatically using

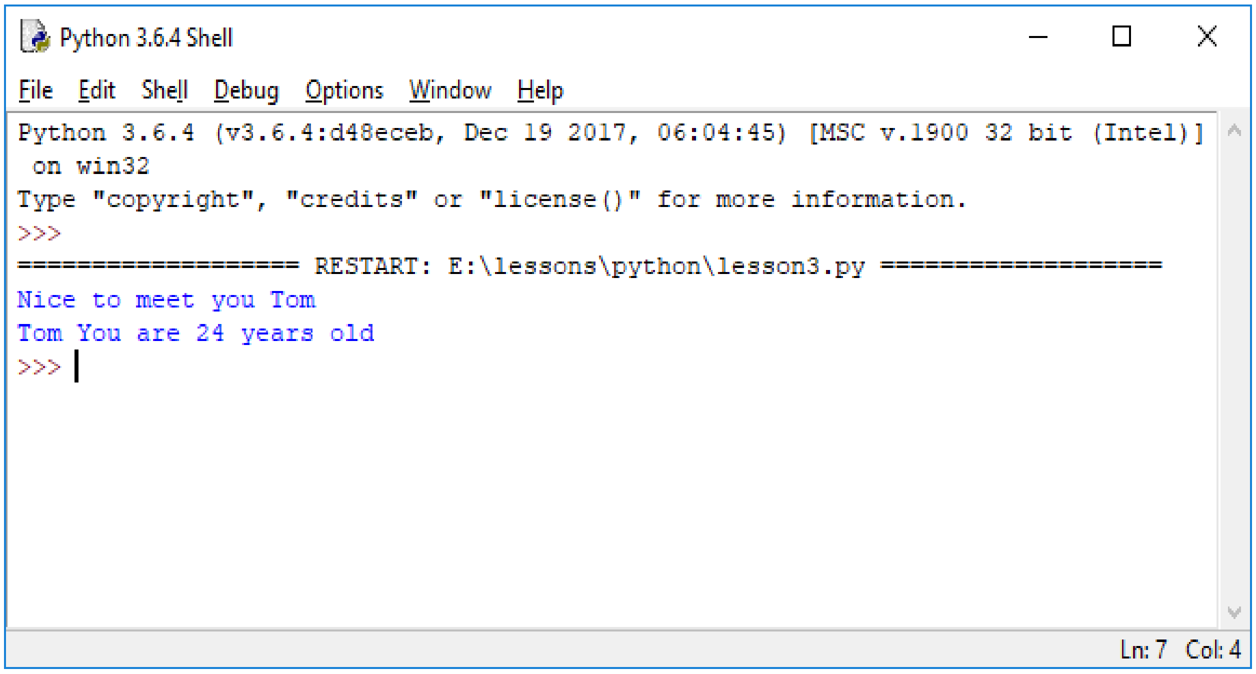
print(p)

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 (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 persons 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 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


# 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 class with the following statement:

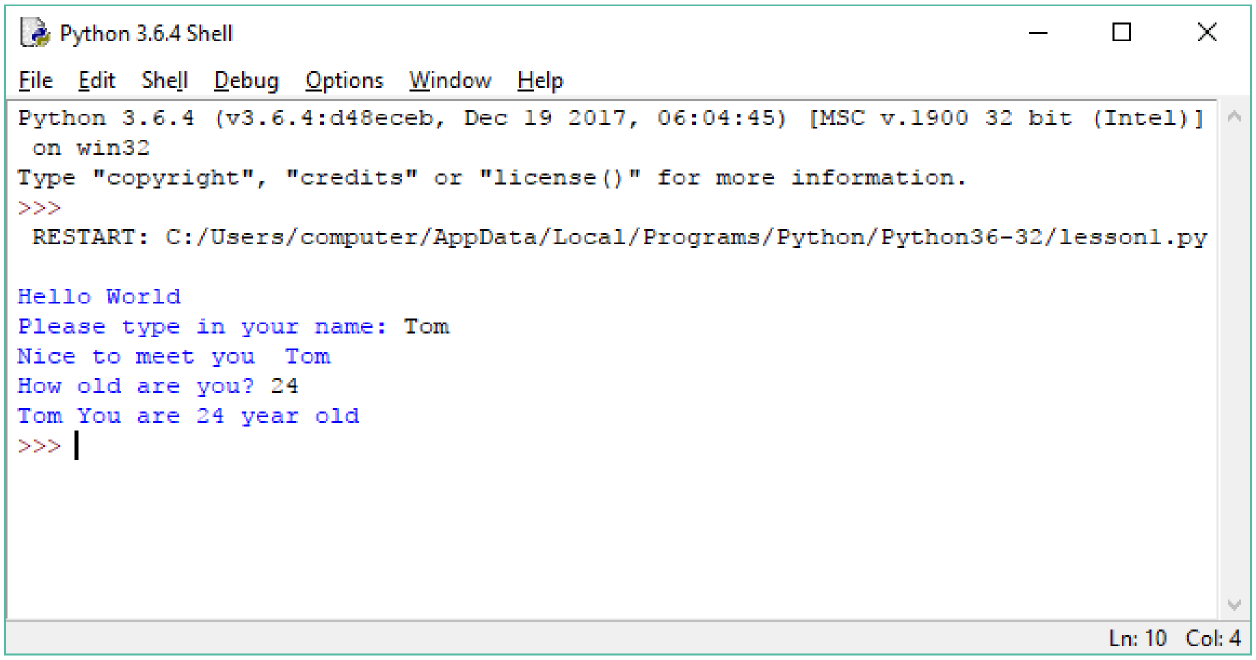
p = Person(name, age)   # create Person class

This statement calls the **__init__()** function of the person class to create the person object and initialized 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 run the program. You will get the same output as the previous program.



```
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
>>> 
```

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
```

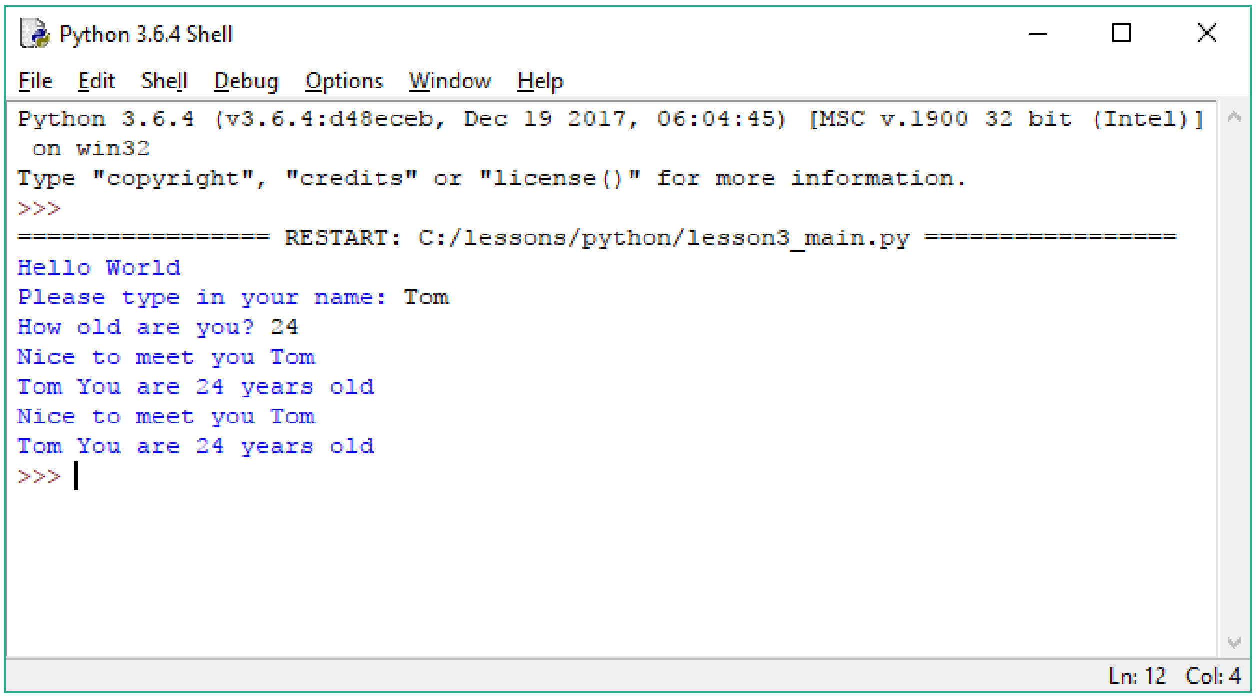
Notice the parameter name is pre-initialized to "" and the parameter age is pre-initialized to 0.

to do:

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

p2 = new Person()

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 (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!

Homework

Take your homework program from Lesson 2 and make a Employee class that has a profession like Doctor, Lawyer, Salesman, Secretary etc. and a salary.

Make constructor _init_() that receives a profession and a salary.
Make getters and setters for profession and salary.
Lastly make a __str__ function use to print out profession and salary.

Make a python file called Homework3.py. You will still need the standalone functions welcome, enterProfession, enterSalary from homework 2.

In the main method make some Employee objects.

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 idnum 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 idnum.

```
# initialize Student
def __init__(self, name, age, idnum):

    Person.__init__(self,name, age)

    self.idnum = idnum
```

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

For new python versions you can call the Person constructor using the super keyword.

```
    # initialize Student
    def __init__(self, name, age, idnum):

        super().__init__(name, age)

        self.idnum = idnum
```

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

```
 # return idnum
def getId(self):

    return self.idnum

# assign idnum
def setId(self,idnum):

    self.idnum = idnum
```

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

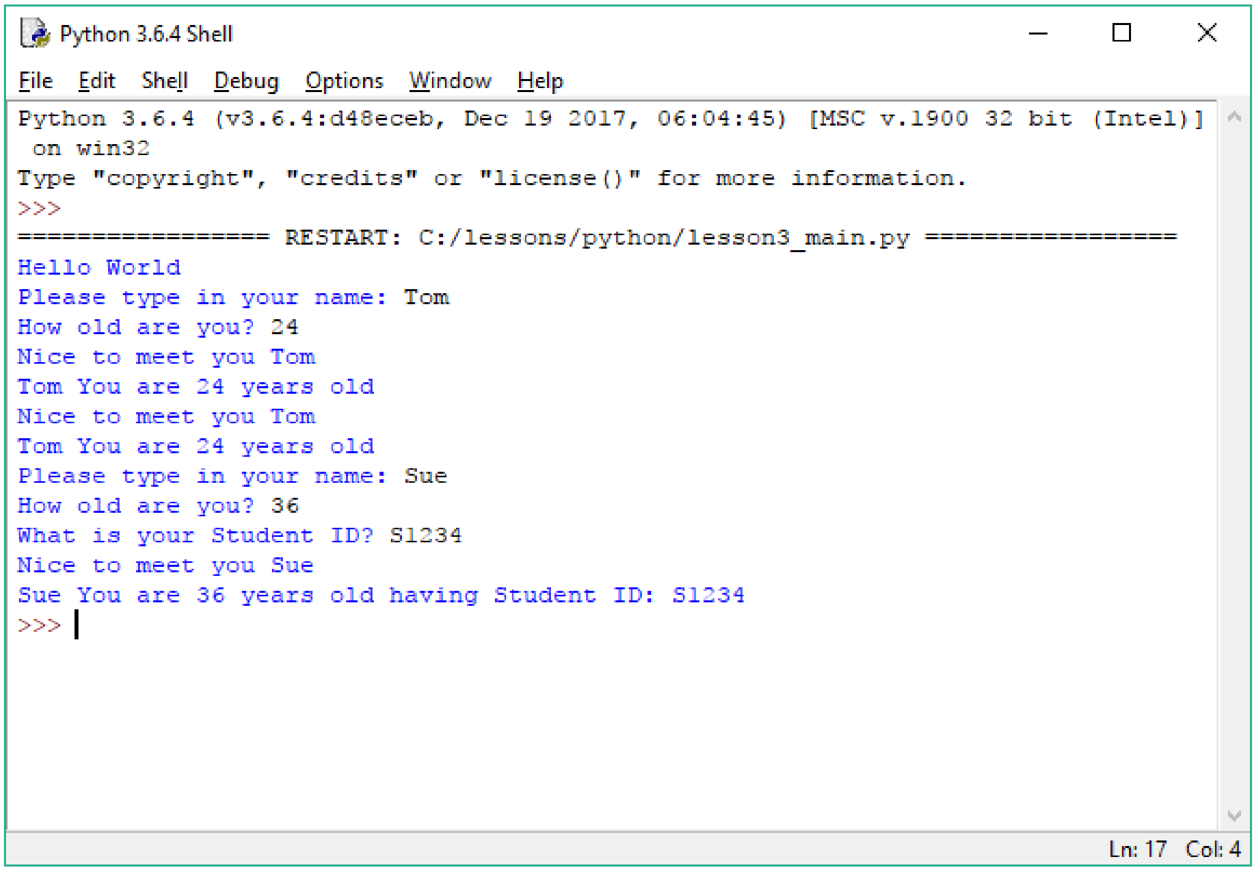
```
    # return student info as a string

      def __str__(self):

        sout = Person.__str__(self)

        sout += " having Student ID: " + self.idnum;

        return sout
```

Alternatively for python 3 you may use the **super** keyword instead.

```
    sout = super().__str__()
```

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

Next make a student object and use the obtained name, age and idnum then print out the student details. You should get something like this:



```
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"):

       Person.__init__(self,name, age)

       self.idnum = idnum
```

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(math.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 them selves. 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.

Homework

Make a class called Manager derived from the Employee class where the manager get a designated bonus. For the Manager class make constructor **__init__** that receives a profession, salary and bonus. The Manager **__init__** function should send the name and age to the Employee **__init__** function. Make getter and setters for the bonus. Finally make a **__str__** function for the Manager class. The **__str__** function should also call the **__str__** function from the Employee class.

The **_str_** function should print out the profession, salary and bonus.

You will also need to make another standalone function enterBonus.

In the main function make some Manager objects. Put every thing in the same file homework3.py used in the previous homework.

Additional homework to do

Put the Employee class and Manager classes in separate files called employee.py and manager.py respectively.

```
from employee import Employee

from manager import Manager
```

In the main file call main using:

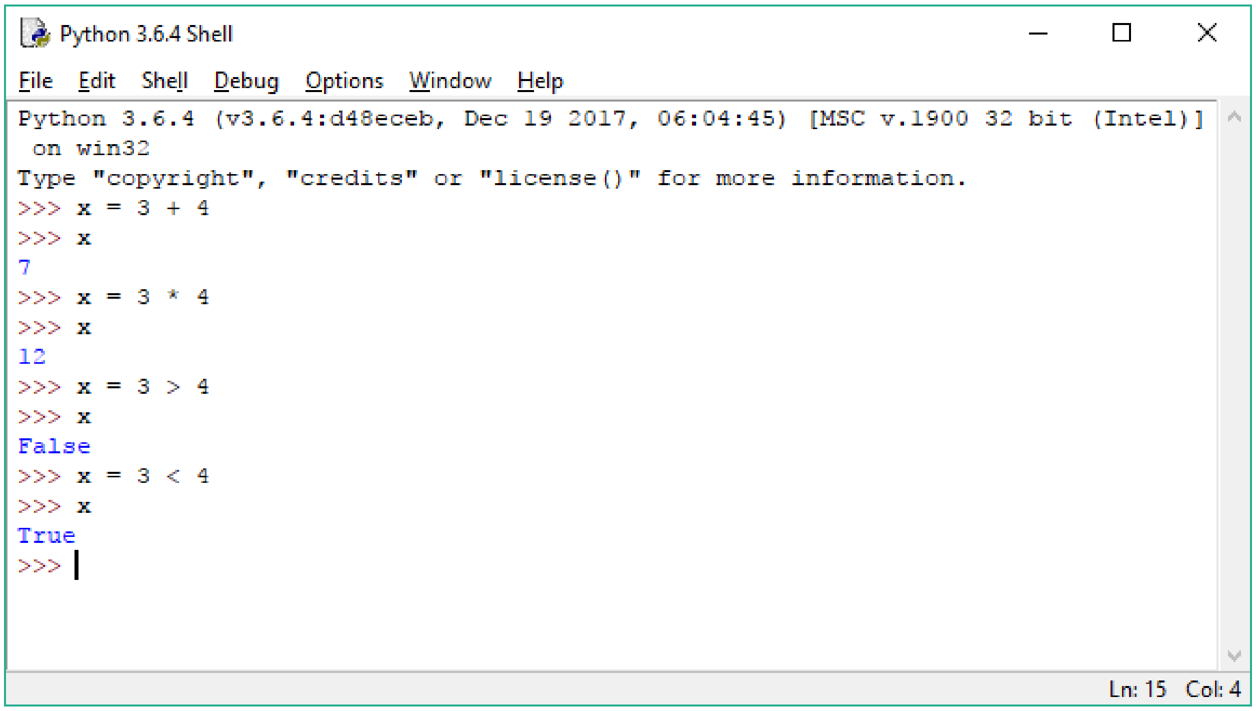
```
if __name__ == "__main__":
   main() # call main function
```

You can still use the same file homework3.py

Lesson 4 Operators, Lists, Sets, Tuples and Dictionaries

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 (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 = 3
y = 4
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 = 3
y = 4
print (x,">",y, " = ",x > y)   # would print  3 > 4 = 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**, ***no*****t** boolean operators.

```
x = True
y = True
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 and 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 NOT reverse bits	~x = -11 (1111 0101)
^	Bitwise XOR same values are 0 opposite vales are 1	x ^ y = 14 (0000 1110)

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'complement to covert 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

x << 1 means to multiply x by 2^1 which is x * 2

x << 2 means to multiply x by 2^2 which is x * 4

x >> 1 means to divide x by 2^1 which is x / 2

x << 2 means to divide x by 2^2 which is x / 4

x = 2
y = 3

print(x,"<<", y,x << y); // 2 << 3 = 8

<<	left shift
	multiply by powers of 2
		x = 5
		x = x<< 2 x * 4 = 20

>>	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 a 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 += 5 = ",x\n",x+=5); // x += 5 = 10

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

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


# 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


# get a substring using slices
# [start index=0 : end index=len()-1 : (optional step=1)]
# (defaults are start and end index)

s4 = s3[0:5]
print (s4)   # hello

s4 = s3[:5]    # uses default start index
print (s4)   # hello

s4 = s3[0:]    # uses default end index's
print (s4)   # hello

s4 = s3[:]    # uses default index's
print (s4)   # hello
```

```
s4 = s3[0:-1]    # print first to last -1 letters
print (s4)   # hell

s4 = s3[:-1]    # print first to last -1 letters  using default start index
print (s4)   # hell

s4 = s3[2:-1]    # print second to last -1 letters  using default start index
print (s4)   # ll

s4 = s3[2:4]    # print second to fourth letters  using default start index
print (s4)   # ll

# add a character to a string
s5 = s3[:5] + 'X' + s3[5:]
print (s5)    # helloXthere

# replace a character in a string

s5 = s5[:5] + '  ' + s5[6:]

print (s5)    # hello there

# reverse a string

s6 = s5[::-1]
print (s6) # erehtXolleh

# make string lower case

s7 = s6.lower() # erehtxolleh
print(s7) # EREHTXOLLEH

# make string upper case

s8 = s6.upper()
print(s8) # 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

# format a string

# method 1

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

print(s9)  # 10.5

#method 2

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

print(s9)    # 10.5

# method 3
x = 10.4564
s9 =  f'${x:.2f}'
print(s9)  # 10.5
```

Homework to do

1. Print out if a number is even, using just a print statement and a arithmetic operator
2. Print out of a number is odd, using just a print statement and a arithmetic operator
3. Swap 2 number using arithmetic operators + and -
4. Swap 2 numbers using bitwise operators xor ^
5. Multiply a number by 8 using a shift operator
6. Divide a number by 8 using a shift operator
7. Make a string and replace the first letter with another letter
8. Make a string and replace the last letter with another letter
9. Make a string and replace the middle letter with another letter
10. Make a string. Split in the middle, swap both parts and reverse the first part.

Call your python file homework4.py

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 lesson4.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

# 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]

# Add a value to a list
list1.append(5)
print(list1)              # [5]

# Print a list
print(list2)              # [1, 2, 3, 4, 5]


# Get a value from a list at a specified location
x = list2[0]
print(x)                  # 1
```

```
# Get part of a list
# [start index=0 : end index=len()-1 : (optional step=1)]
# (defaults are start and end index)
# [:] would there for 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 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]

# Reverse a list
list3 = list2[::-1]
print(list3)                    #[5, 4, 3, 2, 1]

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

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

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

# 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]

# 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']

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

# 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 passed str function, the **map** function converts each number to a string, whereas the **join** function joins all the string numbers together.

```
# list contains a function

def fsq(x):
   return x*x

list1 = [1,2,fsq(3)]

print(list1)  # [1, 2, 9]

# execute a function stored in a list
def  fsq(x):
   return x*x

list1 = [1,2,fsq]
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’]
```

Two Dimensional Lists

Two dimensional lists are analogues to 2 dimensional arrays in other programming languages. To make a two dimensional list 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
```

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 preinitialized 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

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)

# put a string in a set and get individual letters

s=set('tomorrow')    #  {'t', 'w', 'm', 'o', 'r'}

Which is quite different from initializing a set with a string

s = {'tomorrow'}       # {'tomorrow'}

# intersection of 2 sets
 s3 = s1.intersection(s2)
print(s3)                # {5}

# put a set in a list
s1 = {1,2,3}
list1 = [s1]
print(list)      # [{1, 2, 3}]

# put a list in a set
s=set([1,2,3])   # {1, 2, 3}
```

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

To do:

Make a couple of sets of your favourite animals.

Try the other set operations. 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 value from functions or represent multiple values when they are needed. They are read only, meaning you cannot change their values once they are created.

```
# 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)]
```

Dictionaries

Dictionaries contain 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["idnum"]= "S1234"

print(d)   # {'name': 'Tom', 'age': 24, 'idnum': 'S1234'}


# make dictionary with specified keys and values

d2 = {"name": "tom", "age": 24, "idnum": "S1234"}

print (d2);   #  {'name': 'tom', 'age': 24, 'idnum': 'S1234'}


# get values from a dictionary

print(d2["name"])   # tom

print(d2["age"])    # 24

print(d2["idnum"])  # S1234


#print keys of a dictionary

keys = d2.keys()

print(keys)
```

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

```
# print values of a dictionary

values = d2.values()

print(values)
```

```
dict_values(['tom', 24, 'S1234'])
```

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 our next lesson.

To do:

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

Example: fluffy cat

Then make another dictionary d2, use the animal kind as the key and the animal sound as the value.

Example: cat meow

Use the same animal kind's that were in the first dictionary.

Print out the keys of the first dictionary.

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

Get the animal kind from the first dictionary.

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

Fluffy is a cat

From the second dictionary get the sound that animal makes.

Print out what sound animal make's like:

Cat's meow

Call your python file homeworkb.py

Lesson 5   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 lesson5.py to test all the control statements

Branch Control Statements

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
```

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
```

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
```

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")
         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
```

```
5
4
3
2
1
```

for loop

The other loop is the for loop. Ii is much more powerful then the while loop but more difficult to use.

All loops much have counter mechanism. The for loop uses the range function that supplies' the couther stat value step value and end value for the for loop.

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

Statement(s)

The default value for increment 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.

```
# for loop using range
for i in range(1,5+1):
   print (i)
```

```
1
2
3
4
5
```

```
# same as
        for i in range(6):
           print (i)
```

```
1
2
3
4
5
```

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

```
# for loop using range counting backward
for i in range(5,1-1,-1):
   print (i)
```

```
5
4
3
2
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(1,5+1):
   print(r, ":", end="")
   for c in range(1,5+1):
      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
```

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

```
# print out characters in a string
s = "Hello"
for c in s:
   print(c)
```

```
H
e
l
l
o
```

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
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 = {1,2,3,4,5}
for x in set1:
    print (x)
print("")
```

```
1
2
3
4
5
```

Put all letters from a string and put into a set

```
s = "tomorrow"

print(s)
set1 = set()

for c in s:

      set1.add(c)

for x in set1:

    print (x)


print("")
```

```
tomorrow

t
m
o
w
r
```

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

For loops can also print out dictionaries. The main purpose is to print out the dictionary in order by key or order by values

```
# make dictionary
d = {"name": "Bob", "age": 24, "idnum": "S1234"}


# print out dictionary

for key,value in d.items():

   print (key,value)
```

```
name Bob

age 24

idnum S1234
```

```
# print dictionary by key
for key in d.keys() :
        print (key)
```

```
name
age
idnum
```

```
# print dictionary by value
        for key, 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
idnum: Bob
name: s1234
```

We use 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. types. We need to convert all data values to a string data type for sorting. We pass the str faction to the sorted function as the sort key. (sort key is different from dictionary key)

The above loops are 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

Using a comparison function

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


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

print(sort_by_value)
```

The sorted function get a list of dictionary tuple items.

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

The **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 a inline function instead of a separate function. Inline functions are known a 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:

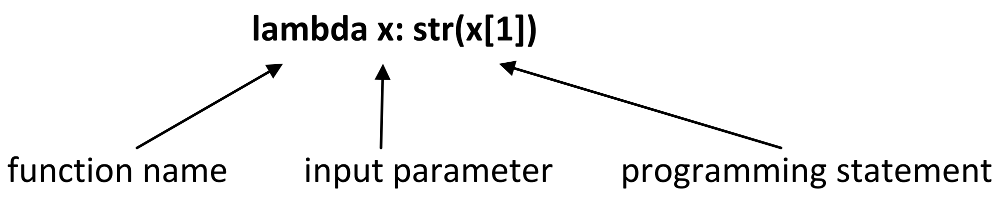
For example:

```
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'),('idnum', 'S1234')]

we are actually sorting the array of dictionary items tuples

dict_items([('name', 'Bob'), ('age', 24), ('idnum', '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.

HOMEWORK

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 tries for each guess and the average tries. Ask the user if they want to play other 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) + 1
```

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 homework5.py or GuessingGame.py

Object Oriented Guessing Game

Make a **GuessGame** class that will keep track of the guess number and tries per game in a list. You should have methods to generate a random number, check if a guess is correct, too low or too high and return the score per game.

The Game __init__ method will generate and store the random number to guess, and make an empty list to store all the user 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 data. The main function would instantiate a new GuesingGame object per game.

After all games have been played print out the tries of each game and the average try per game. Call your python file homework4b.py or GuessingGame2.py
Ask the user if they want to play other game.

LESSON 6 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.

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.close()
```

```
I like programming
```

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 for use the file with 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)                    #    ['I like Programming\n']

# close the file

f.close()
```

readlines actually returns a list of limes. **['I like Programming\n']**

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
```

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

```
# Open the file for read

f = open('input.txt')

# Read the first line


# 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
```

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('input.txt')

# Read the first line

# 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 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,teo,three,four")

f.write("\n")

f.close()
```

Read 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 them 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)

   # printing out individual token
   for t in tokens:

       print(t)

   line = f.readline()

f.close()
```

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

one
two
three
four'
```

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 objects to files using **pickle**.

We first make a book class.

```
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 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
```

Homework To do

Open a text file using try and except that has a small story in it, and count the number of letters, word, sentences and lines. 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 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 homework6.py.

LESSON 7 List Compression, Iterators, Generators and Higher Order Functions

List compression 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, is the best approach to take. It is best to do first, and understand later.

The syntax is very intimidating, but list compression is very power full

new_list = [expression   for loop iteration]

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

This list compression statement is equivalent to:

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

We can expand out 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 compression just to print out even number squares.

new_list = [expression   for loop iteration   filter condition]

The condition is applied to the x in the iteration not to the output value appended to the output list.

```
squares = [x*x for x in range(10) if x % 2 == 0]

print (squares)
```

would print out:

```
[0, 4, 16, 36, 64]
```

To do:

Write a list compression to print out odd number squares

Here is a list compression 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 is a little easier to understand it 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 compression 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 our output. We just print out the even number using a condition filter the condition is applied to the x of the old list not the output o the new list

The syntax is

[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]
```

To do:

Write a list compression to print out the odd numbers.

Compressing tuples

The output list may also contain tuples

```
# 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)]
```

```
# 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)]
```

List compression has many uses like initializing lists with values. Converting values etc. Here are examples to create a 1-dimensional array with list compression

```
oneD_array  = [0 for i in range(3)]
print(oneD_array)
[0, 0, 0]
```

Here are examples to create 2-dimensional array with list compression

```
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 compression

```
mixed_list = ['a',3,'b',5]

sorted_list = sorted([str(x) for x in mixed_list])

print(sorted_list)    #  ['3', '5', 'a', 'b']
```

Using list compression 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']
```

Iterators

Iterators allow you to traverse through a collection value by value. We have already traversed through a list using a for loop. A list is iterable meaning it automatically return an iterator object for traversals.

```
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.

```
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
```

```
1
2
4
9
16
25
36
49
64
81
100
```

```
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.

```
itr = iter(Squares())
for x in itr:
  print(x)
```

Generators

Generators generate values while execution a function and to later retrieve the values after the function has finished executing. The yield statement suspends function's execution and sends a value back to the caller. The function then resumes execution immediately after the last yield run. This allows the function to produce a series of values over time. The values are retrieved from the function after it finishes executing.

Simple generator

The simple generator just returns the value in an iterator after the function finishes executing.

```
def simpleGenerator():
    yield 1
```

The simplegaenerator returns an iterator of values

```
result = simpleGenerator()
```

We iterate through the return iterator and print out the values. In this case it is the value 1

```
for x in result:
    print (x)
```

```
1
```

Square Generator

The square generator uses yield to store the sequence of square values to be retrieved later after the function is called.

```
def squareGenerator():
   for x in range(10):
      yield x*x
```

Here we call the squareGenerator the returns an iterator object.

```
result = squareGenerator()
```

We print out the values from the iterator result in a for loop.

```
for x in result:
   print(x)
```

```
1
2
4
9
16
25
36
49
64
81
100
```

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.

```
zipped = list(zipped)

print(zipped)  # [(1, 'a'), (2, 'b'), (3, 'c')]
```

We then print out the list on separate lines:

```
for x in zipped:

   print(x)
```

```
(1, 'a')
(2, 'b')
(3, 'c')
```

Unzipping a ZIP

We can use list compression to unzip a list:

```
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

We first use the *operator to separate the list into separate arguments'

```
*zipped  #  (1, 'a') (2, 'b') (3, 'c')

unzipped = zip(*zipped)

unzipped = list(unzipped)

print(unzipped)    #   [ (1,2,3) , ('a', 'b', 'c') ]
```

Higher Order Functions

Higher order functions use functions as arguments to an function to do operations on a list using the passed function.

Map

A Map applies a specified function to each item in a list.   The map returns a 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 the 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 items:

   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 an 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 id the programming statement 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:

```
items = [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

Filter

Filter operates similar to map by 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)
```

to do

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.

Use a map, a filter and a lambda function to print out all even square of numbers 1 to 10

Reduce

Reduce takes the first 2 numbers of a list and applies a function to them like adding or subtracting and applies then applies the function to the result and the next number.

list = reduce(f,list)

the function f takes 2 arguments f(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]
list2 = reduce(add, list1)
print(list2)    # 15
```

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 also call reduce   with a lambda

```
list1 = [1, 2, 3, 4, 5])
list2 = reduce((lambda x, y: x + y), list1)
print(list2)   #15
```

You can also use functions that compare values

We can now test if all the numbers are increasing or decreasing in a list

```
list1 = [1, 2, 3, 4, 5])
list2 = reduce((lambda x, y: x > y), list1)
print(list2)  # False
list2 = reduce((lambda x, y: x < y), [1, 2, 3, 4, 5])
print(list2)  # True
```

to do:

apply the reduce function on a function that multiplies 2 numbers together

LESSON 8 RECURSION

When a function calls itself it is known as **recursion**. Recursion is analogues 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();
```

```
I like programming

I like programming

I like programming

I like programming

I like programming
...
```

Unfortunately this program will run forever.

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)
```

It will print I like programming 5 times.

```
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.

Our next example 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.

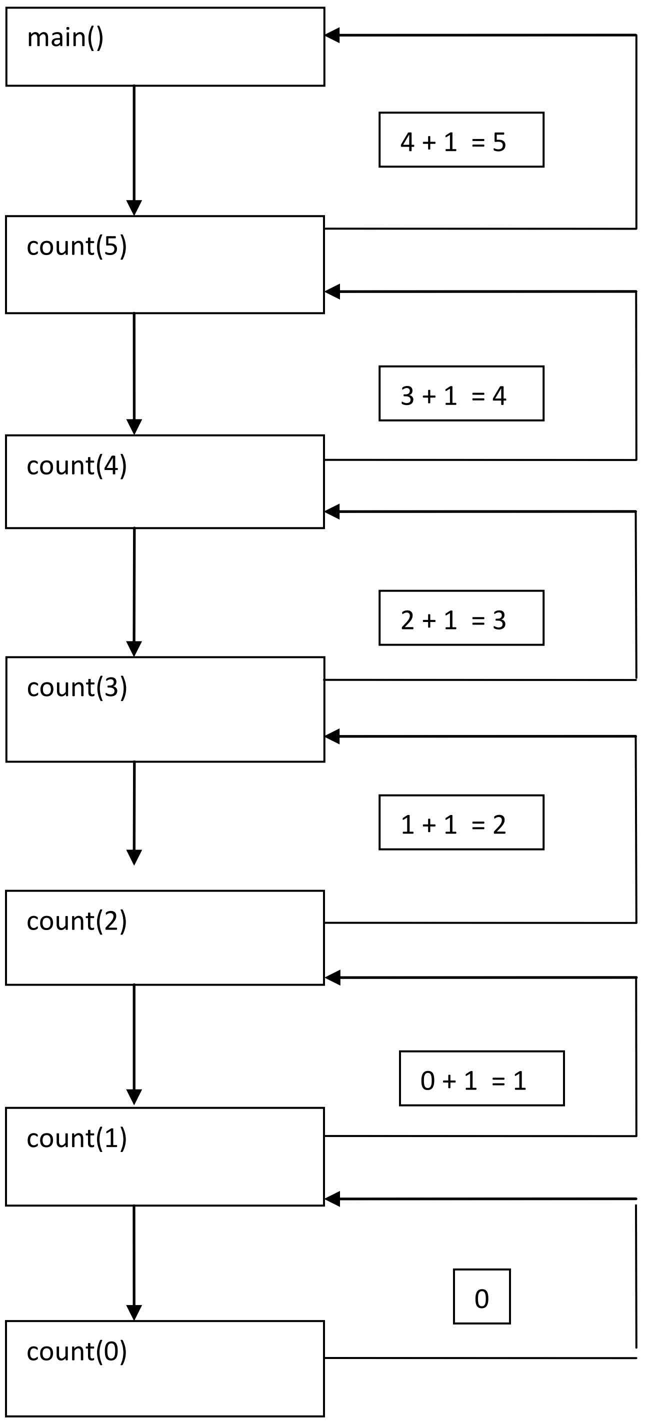
We could rewrite the recursive part as follows:

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

```
def sumn(n):
    if n==0:
        return 0
    else:
        return sumn(n-1) + n
    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 multiples n rather than adding n. This is basically factorial **n.**

```
def multn(n):

    if n==0:

        return 1

    else:

        return multn(n-1) * n;

x = multn(5) # would return 120

print(x) #120

multn(5)
```

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 b 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;
```

x = pown(2,3) # would return 9 because 2*2*2= 8 since 2^3=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;


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 + 1)/2
----------------------
    0          0
    1          1
    2          3
    3          6
    4          10
    5          15
               -----
   Total:      3
```

```
def seqn(n):

    if(n == 0):

        return 0

    else:

        return (n * (n + 1))/ 2 + seqn(n-1)


x = seqn(5) # returns 35

print(x) #35
```

You should print out the individual values of the sequence for n as it calculates the sum of the sequences

```
   x = (n * (n + 1))/ 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 fn of Fibonacci numbers is defined by the recurrence relation.

$f_n = f_{n-1} + f_{n-2}$

with seed values

$f_0 = 0$ 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)


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.

Call it nCr

Pick two letters from set S = {A, B, C, D, E}

Answer:{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

nCr= n! / r!(n-r)!

The Recurrence relation for calculated combinations is:

base cases:

nCn= 1
nC0= 1

recursive case:

nCr= n-1Cr + n-1Cr-1 for n > r > 0

Our recursive function for calculating combinations is:

```
def combinations(n, r):

   if r == 0 or n == r:

        return 1

   else:

        return combinations(n-1, r) + combinations(n-1, r-1)
```

run 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

def print_reverse(s,n):

   if n == 0:

      print("\n")

   else:

      print(s[n-1],end="")

      print_reverse(s, n-1)
```

run like this:

```
s = "tomorrow"

print_reverse(s,len(s))
```

```
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"

```
# 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
```

Combination sets

We have looked at combinations previously where we wrote a function to calculate home 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   5C 2

Answer:{A, B}, {B, C}, {B, D}, {B, E}{A, C}, {A, D}, {A, E}, {C, D}, {C, E}, {D, E}

We are basically filing a seconded 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



s = "ABCDE";
combs = "    ";
```

```
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, combs,start, end, n, r  (5 choose 2)
print_combinations(s, combs,0,len(s)-1,0,2
```

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 miners.

```
# 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

Write a recursive function called **void reverse_string(char*s, int n)** that reverses a string in place. The recursive string receives the string and outputs the string in reverse. No printing is allowed.

Write a recursive function **int search_number(int a[], Int n)** that searched for a number insofar an array and return the index of the number if found otherwise returns -1 if not found.

Write a recursive function **int search_digit(int d, int x)** that searched for a number insofar an number and return 1 of the number if found otherwise returns 0 if not found.

Write a recursive function called **int sum_digits (int d)** that adds up all the digits in a number of any lengths. The recursive function receives an int numbers and returns the sum of all the digits.

Write a recursive function called **void format_number(char* s, int n)** that can insert commas in a number. For example 1234567890 becomes 1,234,567,890

Write a recursive function **int is_even(int n)** that return true if a number has even count of digits or 0 if the number of digits is odd.

Write a recursive function **void print_binary(int 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 is 10. We are going right to left.

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 🡐 1
5/2 = 2
2 %2 = 0   🡐 0
2/2 = 1
1 %2 = 1   🡐 1
1/2 = 0
0 %2 = 0   🡐 0

We are done so going backwards

5 in binary is 0 1 0 1

Write a recursive function **int is_prime(int n)** that returns true (1) if a number is prime otherwise false (0).

A prime number cam only is divides 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 itself. 4 %2 = 0 4 can be divided evenly by 2 so there for 4 is not a prime number.

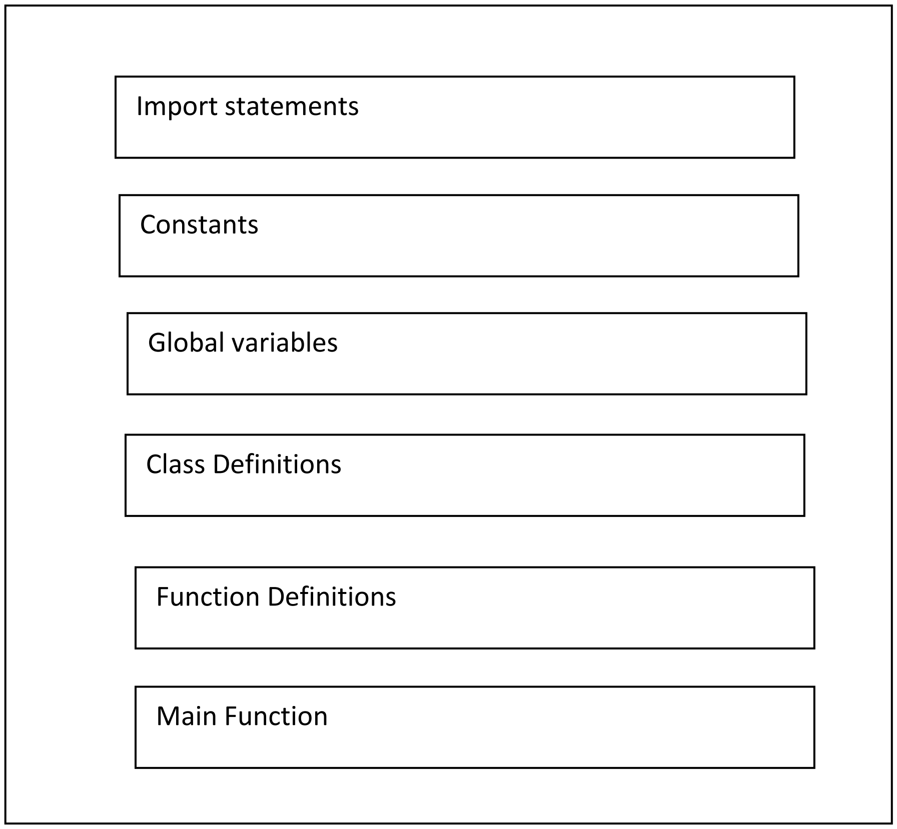
Rewrite isPalindrome using substring slices [:] instead rather than using i and j. Try to ignore spaces, and punctuation.

Put all your functions in a c file called Lesson8.py Include a main function that tests all the recursive functions.

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 store results in a file. 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, scroll up and down contacts as menu operations. Contact names need to be displayed in alphabetically orders order by name. Offer to lookup contacts by name or by phone number. Contacts should be stored in a file, read when app runs, and saved when the app finished running. Save the contacts as csv files. Use a separate function to make a selection menu that returns a user selection. Bonus: View contacts by name or phone number.

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 alphabetically order by name or address. Contacts should be stored in a file, read when app runs, and saved with app finished running. Save the contacts as csv files.

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)
```

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 file, 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

END