1 Introduction to Python

➢ The basic elements of Python

- A Python program, sometimes called a **script**, is a sequence of definitions and commands.
- These definitions are evaluated and the commands are executed by the Python interpreter in something called the shell.
- A new shell is created whenever execution of a program begins.
- In most cases a window is associated with the shell.
- A command called a statement, instructs the interpreter to do something.

For example

The statements print ('*Hello Student, How are you…?*') instructs the interpreter to output the string Hello Student, How are you…? To the window associated with the shell.

The sequence of commands	Interpreter to produce the output
print ('Hello Student!') *print ('Welcome to python class')* *print ('Are you ready?,', 'Lets Start the Python')*	Hello Student! Welcome to python class Are you ready? , Lets Start the Python

- Notice that two values were passed to print in the third statement. The print command takes a variable number of values and prints them, separated by a space character, in the order in which they appear.

❖ Objects, Expressions, and Numerical Types

- Objects are the core things that Python programs manipulate.
- Every object has a type that defines the kinds of things that programs can do with objects of that type.
- Types are either scalar or non-scalar. Scalar objects are indivisible. Think of them as the atoms of the language. Non-scalar objects, for e.g. strings, have internal structure.
- Python has four types of scalar objects:

(1) int

- int is used to represent integers.
- Literals of type int are written in the way we denote integers (e.g., -3 or 5 or 10002).

(2) float

- float is used to represent real numbers.
- Literals of type float always include a decimal point (e.g., 3.0 or 3.17 or -28.72).
- It is also possible to write literals of type float using scientific notation.

- e.g. the literal 1.6E3 stands for 1.6*103, i.e., it is the same as 1600.0.)
- You might wonder why this type is not called real. Within the computer, values of type float are stored in the computer as floating point numbers. This representation, which is used by all modern programming languages, has many advantages.

(3)bool

- bool is used to represent the Boolean values True and False.

(4)None

- None is a type with a single value. We will say more about this when we get to variables.

- Objects and operators can be combined to form expressions, each of which evaluates to an object of some type. We will refer to this as the value of the expression.
  For example,
  Expression 3 + 2 denotes the object 5 of type int,
  Expression 3.0 + 2.0 denotes the object 5.0 of type float.

- The == operator is used to test whether two expressions evaluate to the same value, and the != operator is used to test whether two expressions evaluate to different values.

- The symbol >>> is a shell prompt indicating that the interpreter is expecting the user to type some Python code into the shell. The line below the line with the prompt is produced when the interpreter evaluates the Python code entered at the prompt, as illustrated by the following interaction with the interpreter:

Input	Output
3 + 2	5
3.0 + 2.0	5.0
3 != 2	true

- The built-in Python function type can be used to find out the type of an object:

Input	Output
print(type(3))	<Class int>
print(type(3.0))	<class float>

i+j	If i and j are both of type int, the result is an int. If either of them is a float, the result is a float.
i–j	If i and j are both of type int, the result is an int. If either of them is a float, the result is a float.
i*j	If i and j are both of type int, the result is an int. If either of them is a float, the result is a float.

i//j	If i and j are both of type int, the value of 6//2 is the int 3 and the value of 6//4 is the int 1. The value is 1 because integer division returns the quotient and ignores the remainder.
i/j	In Python 2.7, when i and j are both of type int, the result is also an int, otherwise the result is a float. we will never use / to divide one int by another. We will use // to do that. (In Python 3, the / operator, thank goodness, always returns a float. For example, in Python 3 the value of 6/4 is 1.5.)
i%j	When the int i is divided by the int j. It is pronounced "i mod j," which is short for "i modulo j."
ij**	i raised to the power j. If i and j are both of type int, the result is an int. If either of them is a float, the result is a float.
The comparison operators are == (equal), != (not equal), > (greater),>= (at least), <, (less) and <= (at most).	

❖ Variables and Assignment

- Variables provide a way to associate names with objects. Consider the code

```
pi = 3
radius = 11
area = pi * (radius**2)
print("Area is =",area)
```

Output=Area=363

It first binds the names pi and radius to different objects of type int. It then binds the name area to a third object of type int.

- Consider the two code fragments

```
pi = 3.14
radius = 11.2
area = pi*(radius **2)
print("Area  is =",area)
```

Output=Area=393.8816

It first binds the names pi and radius to different objects of type float. It then binds the name area to a third object of type float.

Typing programs directly into the shell is highly inconvenient. Most programmers prefer to use some sort of text editor that is part of an **integrated development environment** (IDE).

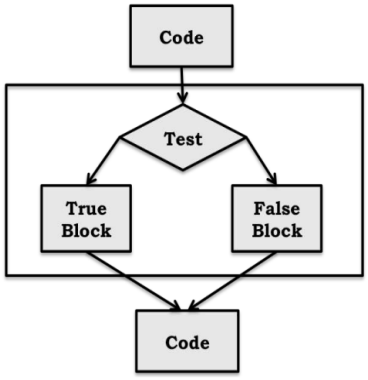
➢ Branching programs

- The kinds of computations we have been looking at thus far are called straight line programs. They execute one statement after another in the order in which they appear, and stop when they run out of statements.

- The kinds of computations we can describe with straight-line programs are not very interesting. In fact, they are downright boring. Branching programs are more interesting. The simplest branching statement is a conditional. As depicted in Figure, a conditional statement has three parts:
- a test, i.e., an expression that evaluates to either True or False;
- a block of code that is executed if the test evaluates to True; and
- An optional block of code that is executed if the test evaluates to False.
- After a conditional statement is executed, execution resumes at the code following the statement.

Syntax:

```
if Boolean expression:
        block of code
else:
        block of code
```



Flow chart for conditional statement

Example:

```
print("Enter any Number = ")
num = int(input())
if (num % 2) == 0:
    print("Number is Even")
else:
    print("Number is Odd")
print("Condtion Complated")
```

Strings

Objects of type str are used to represent strings of characters. Literals of type str can be written using either single or double quotes, e.g., 'abc' or "abc". The literal '123' denotes a string of characters, not the number one hundred twenty-three.

Input	Output
‘a’	a
3*4	12
3*’a’	aaa
3+4	7
‘a’+’a’	aa

The operator + is said to be overloaded: It has different meanings depending upon the types of the objects to which it is applied.

➢ Input

Get input directly from a user, *input* and *raw_input* functions are used. Each takes a string as an argument and displays it as a prompt in the shell. It then waits for the user to type something, followed by hitting the enter key. For *raw_input*, the input line is treated as a string and becomes the value returned by the function; input treats the typed line as a Python expression and infers a type.

```
print("Enter Any Value")
no = input()
print("Enter value  is =",no)
```

➢ Iteration

A generic iteration (also called looping) mechanism is depicted in Figure. Like a conditional statement it begins with a test. If the test evaluates to True, the program executes the loop body once, and then goes back to reevaluate the test. This process is repeated until the test evaluates to False, after which control passes to the code following the iteration statement.

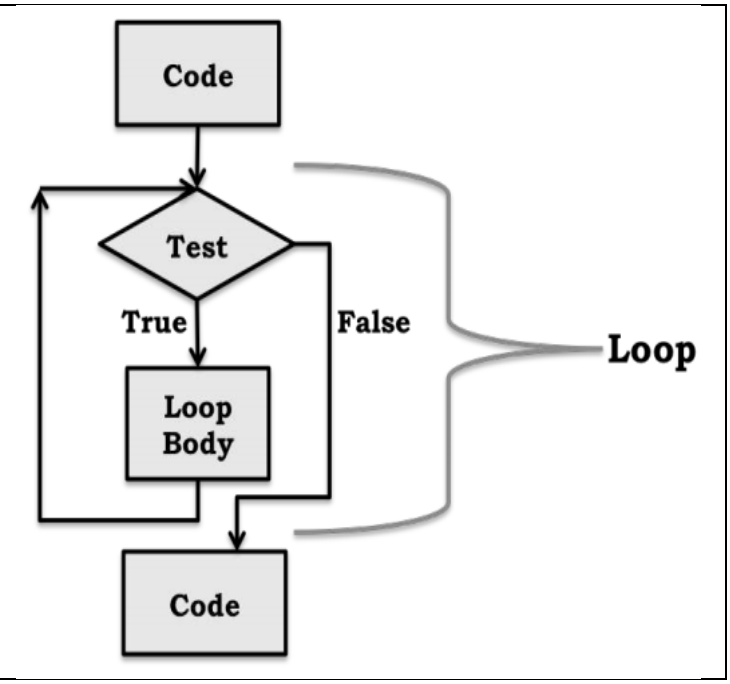
While Loop	For Loop
Syntax **while** ***condition*** **:** ***block of statement***	Syntax ***for variable in sequence:*** ***code block***
Example: a = 0 while a < 10: a = a + 1 print (a)	**Example:** x = 10 for i in range(1,x): print (i)
• The reserved word while begins the while statement. • The *condition* determines whether the body will be executed. A colon (:)	• The variable following for is bound to the first value in the sequence, and the code block is executed. • The variable is then assigned the second value in the sequence, and the code block is executed again. • The process continues until the sequence is exhausted or a break statement is executed within the code block.

Must follow the condition.

• *block* is a block of one or more statements to be executed as long as the condition is true. As a block, all the statements that comprise the block must be indented the same number of spaces from the left. As with the if statement, the block must be indented more spaces than the line that begins the while statement. The block technically is part of the while statement.

#Find the cube root of a perfect cube

```
x = int(input('Enter an integer: '))
ans = 0
while ans**3 < abs(x):
    ans = ans + 1
if ans**3 != abs(x):
    print(x, 'is not a perfect cube')
else:
    if x < 0:
        ans = -ans
    print ('Cube root of', x,'is', ans)
```



➢ Function

In the context of programming, a function is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can "call" the function by name.

Built-in functions

Python provides a number of important built-in functions that we can use without needing to provide the function definition. The creators of Python wrote a set of functions to solve common problems and included them in Python for us to use.

max('Hello world')
min('Hello world')
len('Hello world')

Type conversion functions

Python also provides built-in functions that convert values from one type to another. The int function takes any value and converts it to an integer, if it can, or complains otherwise:

```
int(3.99999)
float('3.14159')
```

Math functions

Python has a math module that provides most of the familiar mathematical functions. Before we can use the module, we have to import it:

User Define Function

We've already used a number of built-in functions, e.g., max and abs. The ability for programmers to define and then use their own functions, as if they were built-in, is a qualitative leap forward in convenience.

Function Definitions

In Python each function definition is of the form

def *name of function (list of formal parameters):*
body of function

- **def** is a reserved word that tells Python that a function is about to be defined.
- **The function name** is simply a name that is used to refer to the function.
- The sequence of names within the parentheses following the function name is the formal parameters of the function.
- When the function is used, the formal parameters are bound (as in an assignment statement) to the actual parameters (often referred to as arguments) of the function invocation (also referred to as a function call).

Example:

```
def mymax(x, y):
    if x > y:
        return x
    else:
        return y

print("Enter any Number = ")
num1 = int(input())
print("Enter any Number = ")
num2 = int(input())
print("max Number",mymax(num1,num2))
```

➢ Scoping

Let Start with example

```
def myfun(x): #name x used as formal parameter
    y = 1
    x = x + y
    print ('x =', x)
    return x

x = 3
y = 2
z = myfun(x) #value of x used as actual parameter
print ('z =', z)
print ('x =', x)
print ('y =', y)
```

When run, this code prints,

```
x = 4
z = 4
x = 3
y = 2
```

What is going on here?

- At the call of myfun(), the formal parameter x is locally bound to the value of the actual parameter x.
- It is important to note that though the actual and formal parameters have the same name, they are not the same variable.
- Each function defines a new name space, also called a scope.
- The formal parameter x and the local variable y that are used in myfun() exist only within the scope of the definition of myfun().
- The assignment statement x = x + y within the function body binds the local name x to the object 4.
- The assignments in myfun() have no effect at all on the bindings of the names x and y that exist outside the scope of myfun().

➢ Specifications

A specification of a function defines a contract between the implementer of a function and those who will be writing programs that use the function. We will refer to the users of a function as its clients. This contract can be thought of as containing two parts:

1. Assumptions: These describe conditions that must be met by clients of the function. Typically, they describe constraints on the actual parameters. Almost always, they specify the acceptable set of types for each parameter, and not infrequently some constraints on the value of one or more of the parameters.

2. Guarantees: These describe conditions that must be met by the function, provided that it has been called in a way that satisfies the assumptions. Functions are a way of creating computational elements that we can think of as primitives. Just as we have the built-in functions max and abs, we would like to have the equivalent of a built-in function for finding roots and for many other complex operations. Functions facilitate this by providing **decomposition** and **abstraction**.

Decomposition creates structure. It allows us to break a problem into modules that are reasonably self-contained, and that may be reused in different settings.

Abstraction hides detail. It allows us to use a piece of code as if it were a black box—that is, something whose interior details we cannot see, don't need to see, and shouldn't even want to see. The essence of abstraction is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context. The key to using abstraction effectively in programming is finding a notion of relevance that is appropriate for both the builder of an abstraction and the potential clients of the abstraction. That is the true art of programming.

➢ Recursion

You may have heard of recursion, and in all likelihood think of it as a rather subtle programming technique. That's an urban legend spread by computer scientists to make people think that we are smarter than we really are. Recursion is a very important idea, but it's not so subtle, and it is more than a programming technique. As a descriptive method recursion is widely used, even by people who would never dream of writing a program.

The world's simplest recursive definition is probably the factorial function on natural numbers. The classic inductive definition is, 1! = 1

(n + 1)! = (n + 1) * n!

Example:

```
def factR(n):
    """Assumes that n is an int > 0 Returns n!"""
    if n == 1:
        return n
    else:
        return n*factR(n - 1)

print("Enter any Number = ")
num = int(input())
print("Factorial=",factR(num))
```

➢ Global variables

If you tried calling function with a large number, you probably noticed that it took a very long time to run. Suppose we want to know how many recursive calls are made? We could do a careful analysis of the code and figure it out. Another approach is to add some code that counts the number of calls. One way to do that uses global variables.

Until now, all of the functions we have written communicate with their environment solely through their parameters and return values. For the most part, this is exactly as it should be. It typically leads to programs that are relatively easy to read, test, and debug. Every once in a while, however, global variables come in handy.

global <Variable Name>

➢ Modules

- We have operated under the assumption that our entire program is stored in one file.
- This is perfectly reasonable as long as programs are small.
- As programs get larger, however, it is typically more convenient to store different parts of them in different files.
- Imagine, for example, that multiple people are working on the same program.
- It would be a nightmare if they were all trying to update the same file.
- Python modules allow us to easily construct a program from code in multiple files.
- A module is a .py file containing Python definitions and statements.
- We could create, for example, a file circle.py containing

```
pi = 3.14159
def area(radius):
return pi*(radius**2)
def circumference(radius):
return 2*pi*radius
def sphereSurface(radius):
return 4.0*area(radius)
def sphereVolume(radius):
return (4.0/3.0)*pi*(radius**3)
```

A program gets access to a module through an import statement. So, for example, the code

```
import circle
print (circle.pi)
print (circle.area(3))
print (circle.circumference(3))
print (circle.sphereSurface(3))
```

- **Files**

Every computer system uses files to save things from one computation to the next. Python provides many facilities for creating and accessing files. Here we illustrate some of the basic ones. Each operating system comes with its own file system for creating and accessing files. Python achieves operating-system independence by accessing files through something called a file handle. The code

```
nameHandle = open('student.txt', 'w')
```

instructs the operating system to create a file with the name student, and return a file handle for that file. The argument 'w' to open indicates that the file is to be opened for writing. The following code opens a file, uses the write method to write two lines, and then closes the file. It is important to remember to close the file when the program is finished using it. Otherwise there is a risk that some or all of the writes may not be saved.

```
nameHandle = open('student.txt', 'w')
for i in range(5):
    print("Enter student name ")
    name = input()
    nameHandle.write(name + '\n')
nameHandle.close()
```

In a string, the character "\" is an escape character used to indicate that the next character should be treated in a special way. In this example, the string '\n' indicates a new line character.

We can now open the file for reading (using the argument 'r'), and print its contents. Since Python treats a file as a sequence of lines, we can use a for statement to iterate over the file's contents.

```
nameHandle = open('student.txt', 'r')
for line in nameHandle:
    print (line)
nameHandle.close()
```

If we don't want to do that we can open the file for appending (instead of writing) by using the argument 'a'.

```
nameHandle = open('kids', 'a')
nameHandle.write('David\n')
nameHandle.write('Andrea\n')
nameHandle.close()
nameHandle = open('kids', 'r')
for line in nameHandle:
print line[:-1]
nameHandle.close()
```

Some of the common operations on files are summarized in below.

open(fn, 'w')	fn is a string representing a file name. Creates a file for writing and returns a file handle.
open(fn, 'r')	fn is a string representing a file name. Opens an existing file for reading and returns a file handle.
open(fn, 'a')	fn is a string representing a file name. Opens an existing file for appending and returns a file handle.
fh.read()	returns a string containing the contents of the file associated with the file handle fh.
fh.readline()	returns the next line in the file associated with the file handle fh.
fh.readlines()	returns a list each element of which is one line of the file associated with the file handle fh.
fh.write(s)	write the string s to the end of the file associated with the file handle fh.
fh.writeLines(S)	S is a sequence of strings. Writes each element of S to the file associated with the file handle fh.
fh.close()	closes the file associated with the file handle fh.

➢ Tuples

Like strings, tuples are ordered sequences of elements. The difference is that the elements of a tuple need not be characters. The individual elements can be of any type, and need not be of the same type as each other.

Literals of type tuple are written by enclosing a comma-separated list of elements within parentheses. For example, we can write

```
t0=(1)
t1 = (1, 'two', 3)
t2 = (t1, 3.25)
print(t0)
print (t1 )
print (t2)
print (t1 + t2)
```

Looking at this example, you might naturally be led to believe that the tuple containing the single value 1 would be written (1). But, to quote Richard Nixon, "that would be wrong." Since parentheses are used to group expressions, (1) is merely a verbose way to write the integer 1. To denote the singleton tuple containing this value, we write (1,). Almost everybody who uses Python has at one time or another accidentally omitted that annoying comma.

➢ Lists and Mutability

Like a tuple, a list is an ordered sequence of values, where each value is identified by an index. The syntax for expressing literals of type list is similar to that used for tuples; the difference is that we use square brackets rather than parentheses. The empty list is written as [], and singleton lists are written without that (oh so easy to forget) comma before the closing bracket. So, for example, the code,

```
L = ['Hello all student', 4, 'Welcome to class']
for i in range(len(L)):
        print (L[i])
```

The output

```
Hello all student
 4
Welcome to class
```

Lists differ from tuples in one hugely important way: lists are mutable. In contrast, tuples and strings are immutable. There are many operators that can be used to create objects of these immutable types, and variables can be bound to objects of these types. But objects of immutable types cannot be modified. On the other hand, objects of type list can be modified after they are created.

```
BCA=['Ashvin', 'Ketan']
BA=['Hitesh', 'Manhar', 'Dixit']
course1 = [BCA, BA]
teacher1 = [['Ashvin', 'Ketan'], ['Hitesh', 'Manhar', 'Dixit']]
print ('course =', course1)
print ('teachers =', teacher1)
print (course1== teacher1)

print (course1 == teacher1) #test value equality
print (id(course1) == id(teacher1)) #test object equality
print ('Id of Course =', id(course1))
print ('Id of Teacher =', id(teacher1))
print(BCA+BA)
```

Notice that the operator + does not have a side effect. It creates a new list and returns it. In contrast, extend and append each mutated contains short descriptions of some of the methods associated with lists. Note that all of these except count and index mutate the list.

L.append(e)	adds the object e to the end of L.
L.count(e)	returns the number of times that e occurs in L.
L.insert(i, e)	inserts the object e into L at index i.
L.extend(L1)	adds the items in list L1 to the end of L.
L.remove(e)	deletes the first occurrence of e from L.
L.index(e)	returns the index of the first occurrence of e in L. It raises an exception if e is not in L.
L.pop(i)	removes and returns the item at index i in L. If i is omitted, it defaults to -1, to remove and return the last element of L.
L.sort()	sorts the elements of L in ascending order.
L.reverse()	reverses the order of the elements in L.

➢ Functions as Objects

In Python, functions are first-class objects. That means that they can be treated like objects of any other type, e.g., int or list.

They have types, e.g., the expression type (fact) has the value <type 'function'>; they can appear in expressions, e.g., as the right-hand side of an assignment statement or as an argument to a function; they can be elements of lists; etc.

Using functions as arguments can be particularly convenient in conjunction with lists. It allows a style of coding called higher-order programming.

```
def factR(n):
    """Assumes that n is an int > 0 Returns n!"""
    if n == 1:
        return n
    else:
        return n*factR(n - 1)
def applyToEach(L, f):
    for i in range(len(L)):
        L[i] = f(L[i])

L = [1, -2, 3.33]
print ('L =', L)
print ('Apply abs to each element of L.')
applyToEach(L, abs)
print ('L =', L)
print ('Apply int to each element of', L)
applyToEach(L, int)
print ('L =', L)
print ('Apply factorial to each element of', L)
applyToEach(L, factR)
print ('L =', L)
```

Output:

L=[1,-2,3.33]
'Apply abs to each element of L.
L = [1, 2, 3.33]
Apply int to each element of L = [1, 2, 3.33]
L = [1, 2, 3]
Apply factorial to each element of L = [1, 2, 3.33]
L = [1, 2, 6]

➢ Strings, Tuples and Lists

We have looked at three different sequence types: str, tuple, and list. They are similar in that objects of all of these types can be operated upon as described in below.

seq[i]	Returns the ith element in the sequence.
len(seq)	Returns the length of the sequence.
seq1 + seq2	Returns the concatenation of the two sequences.
n * seq	Returns a sequence that repeats seq n times.
seq[start:end]	Returns a slice of the sequence.
e in seq is	True if e is contained in the sequence and False otherwise.
e not in seq	Is True if e is not in the sequence and False otherwise.
for e in seq	Iterates over the elements of the sequence.

Some of their other similarities and differences are summarized in below:

Type	**Type of elements**	**Examples of literals**	**Mutable**
String	characters	'', 'a', 'abc'	No
Tuple	any type	(), (3,), ('abc', 4)	No
List	any type	[], [3], ['abc', 4]	Yes

Python programmers tend to use lists far more often than tuples. Since lists are mutable, they can be constructed incrementally during a computation.

➢ Dictionaries

Objects of type dict (short for dictionary) are like lists except that "indices" need not be integers—they can be values of any immutable type. Since they are not ordered, we call them keys rather than indices. Think of a dictionary as a set of key/value pairs. Literals of type dict are enclosed in curly braces, and each element is written as a key followed by a colon followed by a value.

Example:

```
monthNumbers = {'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4, 'May':5,1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May'}
print ('The third month is ' + monthNumbers[3])
dist = monthNumbers['Apr'] - monthNumbers['Jan']
print ('Apr and Jan are', dist, 'months apart')
```

Method	Description
clear()	Remove all items form the dictionary.
copy()	Return a shallow copy of the dictionary.
fromkeys(*seq*[, *v*])	Return a new dictionary with keys from *seq* and value equal to *v* (defaults to None).
get(*key*[,*d*])	Return the value of *key*. If *key* doesnot exit, return *d* (defaults to None).
items()	Return a new view of the dictionary's items (key, value).
keys()	Return a new view of the dictionary's keys.
pop(*key*[,*d*])	Remove the item with *key* and return its value or *d* if *key* is not found. If *d* is not provided and *key* is not found, raises KeyError.
popitem()	Remove and return an arbitary item (key, value). Raises KeyError if the dictionary is empty.
setdefault(*key*[,*d*])	If *key* is in the dictionary, return its value. If not, insert *key* with a value of *d* and return *d* (defaults to None).
update([*other*])	Update the dictionary with the key/value pairs from *other*, overwriting existing keys.
values()	Return a new view of the dictionary's values

Dictionaries are one of the great things about Python. They greatly reduce the difficulty of writing a variety of programs. For example

```
print('Print List when program is starting')
dir1= {"iphone" : 2007,"iphone 3G" : 2008,"iphone 3GS" : 2009,"iphone 4" : 2010,"iphone 4S" : 2011,"iphone 5" : 2012}
print (dir1)
print('Print Add New items')
dir1["iphone 5S"] = 2013
print (dir1)
print('Print Delete  items')
del dir1["iphone"]
print (dir1)
print ('Count number of items',len(dir1))
print ("-" * 10)
print ("iphone List so far: ")
print ("-" * 10)
for d1 in dir1:
   print (d1)
for key,val in dir1.items():
   print (key, "=>", val)
        print ("Print Year:")
        for year in dir1:
           r1= dir1[year]
           print (r1)
```