

FOR BEGINNERS!!

PYTHON

ARCHANA KULKARNI – Year 2015

✹

Table of Contents

GENERIC 2

Assembler 2

Compiler 2

Interpreter 2

Linker 3

Loader 3

Programming language 3

Program 4

Programming 4

Debugging 4

1. Syntax Error 5

2. Run time Error 5

3. Semantic Error 5

Lesson 1- Variables, Expressions & Statements 6

Variable: 6

Expressions: 6

Statements: 6

Lesson 2 8

Lesson 3 13

Lesson 4 25

Lesson 5 30

Lesson 6 38

# GENERIC

## Assembler

A computer will not understand any program written in a language, other than its machine language. The programs written in other languages must be translated into the machine language. Such translation is performed with the help of software called an assembler.

Self/Resident assembler – Runs on a computer and produces the machine codes for the same computer.

Cross Assembler - Runs on a computer and produces the machine codes for other computer.

## Compiler

It is a program, which translates a high-level language program into a machine language program. It goes through the entire program and then translates the entire program into machine codes and hence is slow and occupies a larger part of the memory. A compiler is more intelligent than an assembler. It checks all kinds of limits, ranges, errors etc.

Self/Resident compiler - Runs on a computer and produces the machine codes for the same computer

Cross compiler - runs on a computer and produces the machine codes for other computer.



## Interpreter

It’s a program, which translates statements of a program into machine code i.e. it translates only one statement of the program at a time and executes it. Then it reads the next statement of the program again translates it and executes it, proceeds further till all the statements are translated and executed. A compiler is 5 to 25 times faster than an interpreter

An interpreter is a small program as compared to compiler. It occupies less memory space, so it can be used in a smaller system, which has limited memory space.

Machine codes generated by a compiler are saved permanently for future reference while the machine codes produced by interpreter are not.

**Hence, once the code is compiler it can be executed repeatedly without further translation.**



## Linker

In high-level languages, some built in header files or libraries are stored. These libraries are predefined and these contain basic functions, which are essential for executing the program. These functions are linked to the libraries by a program called Linker. If linker does not find a library of a function then it informs to compiler and then compiler generates an error. The compiler automatically invokes the linker as the last step in compiling a program.

Not only built in libraries, but it also links the user-defined functions to the user-defined libraries. Usually a longer program is divided into smaller subprograms called modules. And these modules must be combined to execute the program, which is done by the linker.

## Loader

Loader is a program that loads machine codes of a program into the system memory.  In Computing, a loader is the part of an Operating System that is responsible for loading programs. Loading a program involves reading the contents of executable file into memory and prepares them for execution.

Once loading is complete, the operating system starts the program by passing control to the loaded program code. All operating systems that support program loading have loaders. In many operating systems the loader is permanently resident in memory.

## Program

It is a sequence of instructions that specifies **how to perform** a computation.

The details look different in different languages but few basic instructions are the same:

1. **Input**: Get data from keyboard or file or any other device
2. **Output**: Send date to screen or a file or any other device
3. **Math**: Perform basic mathematical operations (add, sub, mul, div)
4. **Conditional Execution**: Check for certain condition & execute appropriate code
5. **Repetition**: Perform certain actions repeatedly with some variation

## Programming

It is a process of breaking a lark & complex task into smaller sub-tasks until the sub-tasks are small enough to be performed with one of the above basic instructions.

**Problem Solving** is a process of formulating a problem, finding a solution and expressing a solution.

**Algorithm** is a general process of solving a category of problem

## Programming language

Is needed rather than using spoken language so that there are no ambiguities on the intention or interpretation of what needs to be done by the computer.

**They are also known as formal languages that have been designed to express computations and have strict rules about the syntax.**

Python is an interpreter i.e. it’s a program which taken in some high-level language & interprets that into machine language for the computer.

Basically, the programs that we write run on python interpreter, which in turn runs on the computer.

There are two ways to use an interpreter:

1. **Interactive mode** – We type the python program & the interpreter displays the result.

>>>miles = 26.2 – Assignment & has no visible effect

>>>miles\*1.61 – Expression & has a visible effect & hence we see the result on screen

**42.182**

1. **Script mode** – We type the Python program in a file (called script, file ends with.py) and tell the interpreter to execute the content of the file in which case we’ll have to provide the name of the file.

**Script** - A program stored in a file (usually one that will be **interpreted**)

>>>miles = 26.2

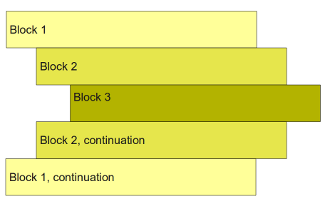
>>>miles\*1.61 – Expression by itself has no visible effect & hence we see no result until we print it out

>>> print miles\*1.61

**42.182**

Unlike other languages, there are no special statement termination characters in python. So, the interpretation here in python is based on the indentation!!!

**Note: Start a python statement with no space.**

****

## Debugging

Programming errors are known as bugs and the process of tracking them down is called as debugging.

Basically, there are 3 kinds of programming errors:

### Syntax Error

Syntax refers to the structure of the program & the rules of that structure.

Ex: Parentheses must appear in pairs, First line of the python program should not have leading spaces etc.

**Python can only execute a program if there are no syntax errors.**

Syntax rules come in two flavors, pertaining to **token** & **structures**.

Token are the basic elements of the language like word or number or symbols and Structure refers to the way the token are arranged

When we read a sentence in a formal language, we will have to figure out what the structure of the sentence is which is known as **parsing** (We do this in natural language but subconsciously)

### Run time Error

These errors refer to the once, which do not show up until the program is up & running. Theses are also called as exceptional errors as they usually indicate that something exceptional (or bad) has happened.

### Semantic Error

Theses errors do not stop the execution of the program as the computer does not generate any errors but the result of running the program would not be the one expected i.e. the program did something other than that was desired.

Theses errors are the most difficult errors to be fixed, as they need you to work backward from the result achieved to where the problem could be.

Ex :

>>>1,00,000

**(1,0,0)->python taken 100000 as comma separated sequence of integers**

# Lesson 1- Variables, Expressions & Statements

## Variable:

A variable by its name is something that can change.

It is a way of referencing a memory location by the computer program i.e. it’s a symbolic name for a physical location. This location holds a value and we/program can use this variable to tell the computer if the value at the location needs to be retrieved or a new value needs to be stored.

Variables in Python –

* Unlike in strongly-types languages such as C or Java where each variable needs to be declared (means binding a variable with a data type) before it is used, there is **no declaration of a variable needed in python**. It’s not even possible.

So, if there is a need of a variable and you have a name thought off, start using the name as a variable.

* Not can the value vary but **even the data type of the variable could change in Python**.

Assignment statement creates a variable & assigns a value to it.

>>> x = 2

>>> y= x

>>> print(**id(x)**)

**4299191744**

>>> print(id(y))

**4299191744**

>>> y=6

>>> print(id(x))

**4299191744**

>>> print(id(y))

**4299191648**

Python chooses a memory location for x and saves the integer value 2. Intuitively, we assume that Python will find another location for the variable y and will copy the value 2 in this place. But Python goes its own way, which differs the ways of C and C++.

As both variables have same value after the assignment, Python lets y point to the memory location of x. >>**4299191744**

Now if Y will be set to a new integer value 6. C programmers will assume, that x will be changed to 6 as well, because we said before, that y "points" to the location of x. But this is not a C-pointer. Because x and y will not share the same value anymore, y gets its own memory location, containing 6 >> **4299191648** (x sticks to 3)

>>>**type**(12)

**<type 'int'>**

>>> type(12.0)

**<type 'float'>**

>>> type('Hi')

**<type 'str'>**

### Naming convention

* Is a combination of letters (upper/lower) & numbers and underscore to separate the word for clarity. General nomenclature is to start a variable name with lower case letter.
* Variable in Python are **case sensitive**

### Python Keywords

Below is a list of **Python keywords**, which belong to building blocks category and the variable names defined by the users, need to be other than these keywords



### Variable Usage

A variable can only be used after it has been defined & not before that. Variable can be constant or be varying.

x = 50 -> constant

x = x + 5 -> varied

## Operators and Operands

Operators are special symbols that represent computations and operands are the value on which the operators get applied or operate.

### Arithmetic

|  |  |  |
| --- | --- | --- |
| **ARITHMETIC OPERATORS (a = 10, b = 20)** | | |
| + | **Addition** - Adds values on either side of the operator | a + b = 30 |
| - | **Subtraction** - Subtracts RHS from LHS | a - b = -10 |
| \* | **Multiplication** - Multiplies values on either side of the operator | a \* b = 200 |
| / | **Division** - Divides LHS by RHS | b / a =2 |
| % | **Modulus** - Divides LHS by RHS and returns remainder | b % a = 0 |
| \*\* | **Exponent** - Performs exponential (power) calculation on operators | a\*\*b = 1020 |
| // | **Floor Division** - The division of operands where the result is the quotient in which the digits after the decimal point are removed. | 9//2 is equal to 4  9.0//2.0 is equal to 4.0 |

### Comparison

|  |  |  |
| --- | --- | --- |
| **COMPARISON OPERATORS (a = 10, b = 20)** | | |
| == | **Equal To** - Checks if the value of two operands are equal or not, if yes then condition becomes true. | (a == b) is False |
| != | **Not Equal To**- Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. | (a != b) is True |
| <> | **Not Equal To**- Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. | (a <> b) is True  This is similar to != operator. |
| > | **Greater Than**-Checks if the value of LHS is greater than the value of RHS, if yes then condition becomes true. | (a > b) is False. |
| < | **Less Than**-Checks if the value of LHS is less than the value of RHS, if yes then condition becomes true. | (a < b) is True. |
| >= | **Greater than or Equal To** -Checks if the value of LHS is greater than or equal to the value of RHS, if yes then condition becomes true. | (a >= b) is False. |
| <= | **Less than or Equal To** Checks if the value of LHS is less than or equal to the value of RHS, if yes then condition becomes true. | (a <= b) is True. |

### Assignment

|  |  |  |
| --- | --- | --- |
| **ASSIGNMENT OPERATOR (a = 10, b = 20, c = 30)** | | |
| = | **Simple assignment** - Assigns values from RHS to LHS | c = a + b 🡪30 |
| += | **Add AND assign**- It adds RHS to the LHS and assign the result to LHS | c += a 🡪 c = c + a 🡪 40 |
| -= | **Subtract AND assign** - It subtracts RHS from the LHS and assign the result to LHS | c -= a 🡪 c = c – a 🡪20 |
| \*= | **Multiply AND assign** - It multiplies RHS with the LHS and assign the result to LHS | c \*= a 🡪 c = c \* a 🡪300 |
| /= | **Divide AND assign** - It divides LHS with the RHS and assign the result to RHS | c /= a 🡪 c = c / a 🡪 3 |
| %= | **Modulus AND assign** - It takes modulus using two operands and assign the result to LHS | c %= a 🡪 c = c % a 🡪 0 |
| \*\*= | **Exponent AND assign** - Performs exponential (power) calculation on operators and assign value to LHS | c \*\*= a 🡪 c = c \*\* a 🡪3010 |
| //= | **Floor Dividion and assign** - Performs floor division on operators and assign value to LHS | c //= a 🡪 c = c // a 🡪3 |

### Bitwise

|  |  |  |
| --- | --- | --- |
| **BITWISE OPERATOR (a = 1010 & b = 1001)** | | |
| & | **Bitwise AND** - Copies a bit to the result if it exists in both operands. | (a & b) 🡪1000 |
| | | **Bitwise OR** - Copies a bit if it exists in either operand. | (a | b) 🡪1011 |
| ^ | **Bitwise XOR** - Copies the bit if it is set in one operand but not both. | (a ^ b) 🡪0011 |
| ~ | **Ones Complement** - is unary and has the effect of 'flipping' bits. | (~a ) 🡪 0101 |
| << | **Left Shift** - The LHS value is moved left by n-number of bits specified by the RHS. | a << 2 🡪1000 |
| >> | **Right Shift** - The LHS value is moved right by n- number of bits specified by RHS. | a >> 2 🡪0010 |

### Logical

|  |  |  |
| --- | --- | --- |
| **LOGICAL OPERATOR (a = True, b = False)** | | |
| and | **Logical AND** - If both the operands are true then the condition becomes true. | (a and b) 🡪 False. |
| or | **Logical OR** - If any of the operands are non-zero then the condition becomes true. | (a or b) 🡪 True |
| not | **Logical NOT** - Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. | not(a and b) 🡪 True. |

### Operator Precedence

The acronym **PEMDAS** is a useful way to remember the rules:

**PEMDAS - P**arentheses, **E**xponentiation, **M**ultiplication, **D**ivision  and Operators with the same precedence are evaluated from left to right

|  |  |
| --- | --- |
| **Operator** | **Description** |
| \*\* | Exponentiation |
| ~ + - | Complement, unary plus and minus |
| \* / % // | Multiply, divide, modulo and floor division |
| + - | Addition and subtraction |
| >> << | Right and left bitwise shift |
| & | Bitwise 'AND' |
| ^ | | Bitwise exclusive `OR' and regular `OR' |
| <= < > >= | Comparison operators |
| <> == != | Equality operators |
| = %= /= //= -= += \*= \*\*= | Assignment operators |
| not or and | Logical operators |

## Expressions

## Statements

## Comments

## Built-in Data Types

* **Arithmetic expression rule:** Expression is something that has a value

Expression -> Expression Operator Expression

Ex:

1. x = 4
2. x= x +10\*50-5

* **Assignment statement rule:**

Variable name = Expression

Ex:

1. variable = 10 + 5
2. name = “archana” + ‘ananth’
3. variable, name = name, variable ->variable = archanaananth, name= 15 i.e. the values are swapped without a temp variable

* **Decimal Number:** It is a fractional number i.e. a number with a decimal point in it.

Ex:

1. 67.56
2. 567.
3. 89.0

* **Strings:** Sequence of characters between quotes (single or double) i.e. the sequence if it starts with a single quote then it ends with a single quote only & if it starts with a double quote then it ends with a double quote only!!

Ex:

1. “hello”,
2. ‘Hi’,
3. ‘How are” you’
4. Print ‘archana’ +’!’\*3 -> archana!!!

Note: In-built function **‘str’** converts anything that is passed to it as a parameters into a string.

Ex: **str(<expression>) -> string** i.e. str(4.5)->4.5 which is a string rather than a numerical value upon which string functions can act on.

* **String indexing:** we can use square brackets to extract part of a string i.e.

<string>[<expression >]. Indexing always start with 0th location in a string.

Ex:

1. ‘Hello’[0] -> H
2. (x+y)[0] -> Concatenates the strings in the variables x, y & extract the 0the element of the resultant string
3. x[0]+y[1] -> Concatenates the 0th element of x & 1st element of y
4. x[-1]-> extracts the last element of x i.e. the characters are count backward(from last) are represented starting from -1
5. ‘archana’[-4]-> h, ‘archana’[3]->h

Note: Length of a string can be found by using the built in function **len(str)**

* **Sub-sequences in a string:** set of charaters in a string starting from a start index till the end index

<string>[: ] -> Extract the entire string

<string>[<expression\_1 > : ] -> extracts the staring starting at expression\_1th location in the string till end

<string>[: <expression\_1>]-> extracts the staring starting at 0th location of the string till expression\_1th location

<string>[<expression\_1 > : <expression\_2>]-> extracts the staring starting at expression\_1th location in the string till expression\_2th location

Ex:

name = ‘aarush’

1. name[:] ->aarush
2. name[3:] ->ush
3. name[:3] ->aar
4. name[2:6]->rush
5. name[2:2]-> empty string as there is no characters between the indexes
6. name[:-1]->aarus

Note: You can read a string backwards with the following syntax:

**<string>[::-1**] - where the "-1" means one step back

* **Finding strings in strings:** Python has a build in procedure called ‘Find’ which can be used to find a string with another string.

**<string>.find(<sub\_string>)** results in the first position/location (is a number/index) where the sub\_string appears with in the string and results -1 when the sub\_string is not foung in the string.

Ex:

1. ‘text’.find(‘t’) ->0
2. “test”.find(“st”)->2
3. ‘Test’.find(‘t’)-> -1
4. ‘test’.find(‘test’+’!!’)+1 -> 0
5. ‘string’.find(‘’)-> 0, always be zero no matter what the string is as be are trying to find an empty string

**<string>.find(<sub\_string>, <expression>)** result in the first position/location (is a number/index) where the sub\_string appears with in the string starting from the expressionth position in the string.

Ex: string = ‘Udacity’ , sub\_string = ‘city’

1. string.find(sub\_string,0)->3 i.e. search starts at 0th(Udacity) location
2. string.find(sub\_string,3)->3 i.e. search starts at 3rd(city) location
3. string.find(sub\_string,4)->-1 i.e. search starts at 4th(city) location
4. string[3:].find(sub\_string)->0 i.e search starts at 0th location of string ‘city’
5. string[3:].find(sub\_string[3:])->3 i.e search starts at 0th location of string ‘city’ for the sub-string ‘y’
6. **Finding URL in a page**

page = “… content of a web page…”

start\_link = page.find(‘<a href =’)

start\_quote = page.find(‘”’,start\_link)

end\_quote = page.find(‘”’,start\_link+1)

URL = page[start\_quote+1 : end\_quote]

# Lesson 2

* **Procedure**: Is a set of instructions which take in parameters, act upon then as per the instructions & return the result as an output

**def <proc\_name>(<parameters>)**:

**<Block\_start>**

.

.

**<Block\_end>**

……..

Note: ‘def’ stands for definition followed b y a procedure name which follows the same naming convention as a variable, then we have list of names/parameters each one separated by a comm. These act as input to the procedure. Colon (: marks the beginning of procedural steps that act on the input. First statement in the block is indented in a particular fashion (4 white spaces usually)which is followed till the last statement. Block\_end most often is “return <expression1>, <expression2>…”

Ex:

**Def sum(a,b):**

**a= a+b**

**# return a**

**print sum(2,3)**

🡪 results in printing 5 if the return statement is uncomment else prints “None” since the function has no output(return value)

* **Procedure composition:** This means that the output of one procedure is taken as an input of another procedure

Ex: print **sum(sum(2,3),5)** -> 10

* **Conditional statements:** allows us to make decision based on the result of a condition or an expression.

**If <test\_expression>:**

**<True\_Block>**

**else:**

**<False\_Block>**

* **Boolean Operators:** Two are more operands are operated using the Boolean operators (and, or, not) to result in a Boolean value (True/False)
* **Comparison operators:** Two operands are operated using the comparison operators (>, <, !=, ==, >=, <=) to result in a Boolean value
* **While Loops:** True\_block is executed as long as the test\_expression results in True

**while <test\_expression>:**

**<True\_Block>**

**<False\_Block>**

Note: Factorial (n\*n-1\*n-2\*…\*1) is account to number of combinations possible with n number of items. Example could be number of ways to arrange 52 different cards (52!)

* **Break Statement:** This gives us the flexibility to exit out of a loop when a condition is true

**while <test\_expression>:**

**<Start\_Block>**

**…**

**…**

**if <test\_expression>:**

**break**

**…**

**…**

**<End\_Block>**

* **Your age in days code**

**def isLeapYear(year):**

if (year % 400 ==0):

return True

if (year % 100 ==0):

return False

if (year % 4 ==0):

return True

return False

**def dayOfMonth(year,month):**

if(month == 1 or month ==3 or month == 5 or month == 7 **\**

or month ==8 or month ==10 or month == 12):

return 31

else:

if(month == 2):

if (isLeapYear(year)):

return 29

else

return 28

return 30

**def nextDay(year, month, day):**

"""Simple version: assume every month has 30 days"""

if day < dayOfMonth(year,month):

return year, month, day + 1

else:

if month == 12:

return year + 1, 1, 1

else:

return year, month + 1, 1

**def dateIsBefore**(year1, month1, day1, year2, month2, day2):

"""Returns True if year1-month1-day1 is before year2-month2-day2. Otherwise, returns False."""

if year1 < year2:

return True

if year1 == year2:

if month1 < month2:

return True

if month1 == month2:

return day1 < day2

return False

**def daysBetweenDates**(year1, month1, day1, year2, month2, day2):

"""Returns the number of days between year1/month1/day1

and year2/month2/day2. Assumes inputs are valid dates

in Gregorian calendar."""

# program defensively! Add an assertion if the input is not valid!

**assert** **not** dateIsBefore(year2, month2, day2, year1, month1, day1)

days = 0

while dateIsBefore(year1, month1, day1, year2, month2, day2):

year1, month1, day1 = nextDay(year1, month1, day1)

days += 1

return days

def **test():**

test\_cases = [((2012,1,1,2012,2,28), 58),

((2012,1,1,2012,3,1), 60),

((2011,6,30,2012,6,30), 366),

((2011,1,1,2012,8,8), 585 ),

((1900,1,1,1999,12,31), 36523)]

for (args, answer) in test\_cases:

result = daysBetweenDates(\*args)

if result != answer:

print "Test with data:", args, "failed"

else:

print "Test case passed!"

test()

-------------------------------------------------------------------------------------------

**def isLeapYear(year):**

if ((year % 400 == 0) **or** ((year % 4 == 0) **and** (year % 100 != 0)):

return True

return False

**def daysTillNow(day,month,year):**

**daysOfMonths** = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

if(isLeapYear(year)):

daysOfMonths[1] += 1

days = ((year-1)\*365) + ((year-1)/4 + (year-1)/400 - (year-1)/100) +

**sum**(daysOfMonths[0:month-1]) + day

return days

**def daysBetweenDates(year1, month1, day1, year2, month2, day2):**

return **abs**(daysTillNow(day1,month1,year1) - daysTillNow(day2,month2,year2))

* **ABACUS code:**

**def print\_abacus(value)**

text = **str**(value)

length = **len**(text)

left = 10 -length

i = 0

while(left > 0):

print '|' + '0'\*5 +'\*'\*5 + ' ' + '|'

left -= 1

while(i< length):

n = **int**(text[i])

if (n <= 5):

print '|' + '0'\*5 +'\*'\*(5-n) + ' ' + '\*'\*n + '|'

else:

print '|' + '0'\*(10-n) + ' ' + '0'\*(n-5) + '\*'\*5 + '|'

i += 1

------------------------------------------------------------------------------------------------------

**def print\_abacus(value):**

text = '0'\*(10 - len(str(value))) + str(value)

i = 0

while(i < 10):

print '|00000\*\*\*\*\*|'[0:11-int(text[i])] + ' ' + '|00000\*\*\*\*\*|'[11- (int(text[i])):]

i += 1

# Lesson 3

* **Structured data: “Strings”** are atype of structured data types as we can extract the individual characters of the string & operate on the sub-sequence strings too. “Lists” are one another kind of structured data where in it can be a sequence of anything (characters/numbers or other lists)

**Ex of lists:**

1. List1 = [‘h’, ‘e’, ‘l’, ‘l’, ‘o’]
2. List2 = [1,2,3,4,5]
3. List3 = [[‘India’, ‘Hindi’, ‘Hinduism’],

[‘USA’, ‘English’, ‘Christianity’],

[‘Chine’, ‘Chinese, ‘Buddhism’]]

Note: Like the strings, lists are also indexed starting from 0th location.

* **Mutation:** This is the major difference between a string(or even number) & a list. Mutation refers to a permanent change to the object. A string is a non-mutable object while a list is a mutable object

**Ex :**

* 1. **Strings are non-mutable:**

name = ‘archanb’

name = ‘Archana’

name = ‘Archana’ + ‘Ananth’

In the 3 cases above, value of variable ‘name’ is modified however the string itself is not, not even in the 3rd case on concatenation

* 1. **Numbers are non-mutable**

num=5

def inc (a):🡪 refers to same location as ‘num’

a +=1 🡪 refers to a new location which has an incremented value as the numbers are

non-mutable

print a -------🡪 6

inc(num)

print num----------🡪5

* 1. **Lists are mutable:**

name = [‘a’, ‘a’, ‘r’, ‘u’, ‘s’, ‘h’]

name [0] = ‘A’ 🡪Aarush

Since the lists are mutable objects, we will have to take care of all the other objects that refer to the same list before modifying it.

num=[5,6,7]

def inc (a):

a[2] +=1 🡪 still refers to the location which ‘num’ points to but with an updated values

as the lists are mutable

print a -------🡪 [5,6,8]

inc(num)

print num----------🡪[5,6,8]

----------------------------------------

num = [1,2]

def inc (n):

**n.append(3)**

print n

inc(num)

print num

Output: [1,2,3]

[1,2,3]

-----------------------------------

num = [1,2]

def inc (n):

**n = n.append(3)**

print n

inc(num)

print num

Output: **none**

[1,2,3]

-----------------------------------

num = [1,2]

def inc (n):

**n += [3]**

print n

inc(num)

print num

Output: [1,2,3]

[1,2,3]

----------------------------------------

num = [1,2]

def inc (n):

**n = n +[3]**

print n

inc(num)

print num

Output: [1,2,3]

**[1,2]**

* **Aliasing:** When two names refer to a same object that means the object has an alias/alternate name
* **List Operation:** There are 3 main operators that are most commonly used with list.

1. **append:** A method which appends an character/number/list at the end of a list.

**Ex:**

* 1. list1 = [1,2,3]

list1.append(4) -🡪 [1,2,3,4]

* 1. list2 = [‘H’, ‘I’]

list2.append(‘!’) 🡪[‘H’, ‘I’, ‘!’]

* 1. list1.append(list2) 🡪[1,2,3,4,[‘H’, ‘I’, ‘!’]]

1. **concatenate(+)** : Concatenates two lists

**Ex**:

* 1. list1 = [1,2]

list2 = [‘H’, ‘I’]

list = list1 + list2 🡪 [1,2,’H’,’I’]

* 1. list1 = [1,2]

list2 = list1

list1 = list1 + [3] 🡪 list1 points to a new list [1,2,3] but the original list [1,2] remain unchanged

print list1, list2 🡪 list2 is [1,2]

1. **len:** Provides the number of elements in the list. This only count the outer most elements of the list.

**Ex:**

* 1. list1 = [1,2,3]

len(list1) 🡪 3

* 1. string = ‘Hello’

len(string) 🡪 5

* 1. list2 = [5,6]

len(list1.append(list2)) 🡪 i.e [1,2,3,[5,6]] 🡪 4

* **Data Stores:** "memory" is usually semiconductor storage read-write random- access memory, typically DRAMs (Dynamic-RAM, volatile) or other forms of fast but temporary storage. "Storage" consists of storage devices and their media not directly accessible by the CPU (secondary or tertiary storage), typically hard disk drives, optical disc drives, and other devices slower than RAM but non-volatile (retaining contents when powered down)

1. Registers: part of the processor. Fasted memory that can be accessed by CPU. Expensive & volatile. Their latency is around 0.4 nSec( 1/109 sec).
2. DRAMS (capacitor): They are volatile but faster than secondary memory. Their latency is around 12 nSec. Capacity is in terms of GigaBytes (230 bytes). 2GB DRAm costs around $10
3. Hard Drive: They are non-volatile but slower than the primary memory. Their latency is around 7 mSec. Capacity is in terms of TeraBytes (240 bytes). 1TB hard drive costs around $100

Note:

1. **Bandwidth (computing)**: rate of data transfer, bit rate or throughput, measured in bits per second (bit/s)
2. **Latency:** time interval between the stimulation and response, or, from a more general point of view, as a time delay between the cause and the effect of some physical change in the system being observed.
3. **cyclic redundancy check (CRC):** an error detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data. Blocks of data entering these systems get a short check value attached, based on the remainder of a polynomial division of their contents; on retrieval the calculation is repeated, and corrective action can be taken against presumed data corruption if the check values do not match.

**Ex:** 14 bits of message with a 3-bit CRC, with a polynomial x³+x+1

11010011101100 000 <- input right padded by 3 bits

1011 <- divisor

01100011101100 000 <- result (first four bits are the XOR with the divisor beneath, the rest of the bits are unchanged)

1011 <- divisor ...

00111011101100 000

1011

00010111101100 000

1011

00000001101100 000 <- divisor moves over to align with the next 1 in the dividend (as quotient for that step was zero)

1011 (in other words, it doesn't necessarily move one bit per iteration)

00000000110100 000

1011

00000000011000 000

1011

00000000001110 000

1011

00000000000101 000

101 1

-----------------

00000000000000 100 <--- remainder (3 bits). Division algorithm stops here as quotient is equal to zero.

The validity of a received message can easily be verified by performing the above calculation again, this time with the check value added instead of zeroes. The remainder should equal zero if there are no detectable errors.

11010011101100 100 <--- input with check value

1011 <--- divisor

01100011101100 100 <--- result

1011 <--- divisor ...

00111011101100 100

......

00000000001110 100

1011

00000000000101 100

101 1

------------------

0 <--- remainder

* **For Loop:** This control structure is useful when we want to iterate through all the elements of a list.

**for <name> in <list>:**

**<Block>**

On each iteration, an element from the list is assigned to a new variable <name> which can then be used if needed to execute the <Block>. This continues until all the elements in the lists are evaluated.

**Ex:**

def **find\_element**(list,value):

count = 0

**for** element **in** list:

if (element == value):

return count

count += 1

return -1

**print find\_element([1,2,3],3)**

Note: <name> is just an element in the list but this is not a <text\_expression>

* **Index list operation:** This method operates on a list & return the first index location where <value> is found but return an error if the <value> is not found in the list.

<list>.**index**<value>

* **In list operation (not a method):** This operator goes through each element of the list to find if the <value> passed in is found in the list or not. Returns true if the <value> is found else returns false.

<value> **in** <list>

<value> **not** **in** <list> 🡪 return values are exactly opposite to “in” operator

**Ex:**

def **find\_element**(list,value):

if(value **in** list):

return list.**index**(value)

return -1

**print find\_element([1,2,3],3) -🡪2**

**-----------------------------------------------------**

def **union**(list1,list2):

**for** index **in** list2:

if(index **not in** list1):

list1.**append**(index)

a = [1,2,3]

b = [2,4,6]

**union(a,b)**

**print a** 🡪 [1,2,3,4,6]

**print b** 🡪[2,4,6]

* **Pop list operation:** This method pulls out the last element of the list i.e. it return the last element of the list and also mutates the original list by deleting the popped out element from it.

**Ex:**

list1 = [1,2,3]

list2 = []

list2 = list1.pop

🡪 list1 = [1,2]

list2 = [3]

**Note**: By default the pop functions pulls out the last element in the list i.e right most element but when a parameter passed, pop function pulls out the element at the location passed as parameter from the list

**List1.pop(0) >> 1**

* **Web Crawler:**

**def get\_page(url):**

# This is a simulated get\_page procedure so that we can code on two pages "http://xkcd.com/353" and "http://xkcd.com/554".

**try**: #this is used as an error proof, we try what is under “try” block, if none of then work out then we execute the “except” block

**if url == "http://www.udacity.com/cs101x/index.html":**

return ('<html> <body> This is a test page for learning to crawl! '

'<p> It is a good idea to '

'<a href="http://www.udacity.com/cs101x/crawling.html">learn to '

'crawl</a> before you try to '

'<a href="http://www.udacity.com/cs101x/walking.html">walk</a> '

'or <a href="http://www.udacity.com/cs101x/flying.html">fly</a>. '

'</p> </body> </html> ')

**elif url == "http://www.udacity.com/cs101x/crawling.html":**

return ('<html> <body> I have not learned to crawl yet, but I '

'am quite good at '

'<a href="http://www.udacity.com/cs101x/kicking.html">kicking</a>.'

'</body> </html>')

**except:**

return ""

return ""

def **get\_next\_target**(page):

start\_link = page.find('<a href=')

if start\_link == -1:

return None, 0

start\_quote = page.find('"', start\_link)

end\_quote = page.find('"', start\_quote + 1)

url = page[start\_quote + 1:end\_quote]

return url, end\_quote

def **get\_all\_links**(page):

links = []

while True:

url,endpos = get\_next\_target(page)

if url:

links.append(url)

page = page[endpos:]

else:

break

return links

def **union**(p,q):

for e in q:

if e not in p:

p.append(e)

def **crawl\_web**(seed):

tocrawl = [seed]

crawled = []

while tocrawl:

page = tocrawl.**pop**()

if page not in crawled:

crawled.**append**(page)

**union**(tocrawl,**get\_all\_links(get\_page(page))**)

return crawled

print crawl\_web("http://www.udacity.com/cs101x/index.html",0)

---------------------------------------------------------------------------------------------------

Modify the crawl\_web procedure to take a second parameter, max\_depth, that limits the depth of the search. We can

define the depth of a page as the number of links that must be followed to reach that page starting from the seed page, that is, the length of the shortest path from the seed to the page. No pages whose depth exceeds max\_depth should be included in the crawl.

def **crawl\_web**(seed,max\_depth):

tocrawl = [seed]

crawled = []

**next\_depth = []**

depth = 0

while (tocrawl and **depth <= max\_depth):**

page = tocrawl.pop()

if page not in crawled:

union(**next\_depth**, get\_all\_links(get\_page(page)))

crawled.append(page)

if (not tocrawl):

**tocrawl, next\_depth = next\_depth,[]**

depth += 1

return crawled

* **Sudoku:** whole numbers from 1 to n exactly once

def **check\_sudoku(sud):**

n = len(sud)

shiftD = 0

**for row in sud:**

shiftR = 0

shiftD += 1

**for col in row:**

if (**col > n or col <= 0**): # numbers from 1 to n

return False

elif(**'.' in str(col)): #**decimal numbers not allowed

return False

else:

c, r = 0, 0

shiftR += 1

while(c+shiftR < n):

if(col == row[c+shiftR]):

return False

c += 1

while(r+shiftD < n):

if(col == sud[r+shiftD][shiftR-1]):

return False

r += 1

return True

------------------------------------------------------------------------------------------

def **check\_sudoku(matrix):**

n = len(matrix)

digit = 1

**while(digit <= n):** # for each digit

row = 0

while(row < n): # for each row and column at the same time

col = 0

row\_count, col\_count = 0, 0

while(col < n): # for each element in row/column

if(matrix[row][col] == digit):

row\_count += 1

if(matrix[col][row] == digit):

col\_count += 1

col +=1

**if (row\_count != 1 or col\_count != 1):**

return False

row += 1 #next row/col

digit += 1 #next digit

return True

* **Symmetric square list:** 1st row is the same as the 1st column, 2nd row is the same as the 2nd column and so on.

def **symmetric(matrix):**

n = len(matrix)

row = 0

**for e in matrix:** # square list or not

if(len(e) !=n):

return False

while(row < n):

col = 0

while(col < n):

if(**matrix[row][col] != matrix[col][row]**): # symmetrix or not

return False

col += 1

row += 1

return True

* **Numbers in lists:** Every number in the string should be inserted into the list. If a number x in the string is less than or equal to the preceding number y, the number x should be inserted into a sublist. Continue adding the following numbers to the sublist until reaching a number z that is greater than the number y. Then add this number z to the normal list and continue

def **numbers\_in\_lists(string):**

n = len(string)

list = []

sub\_list = []

list.append(int(string[0])) # convert the string into integer

i, j = 1, 1

while(i < n):

if(string[i] <= string[j-1]):

**sub\_list.append(int(string[i]))**

else:

if(**sub\_list):** # if the sub list is not empty

list.append(sub\_list)

list.append(int(string[i]))

j = j+ len(sub\_list) +1

sub\_list = []

i += 1

if(sub\_list):

list.append(sub\_list)

return list

# Lesson 4

* **Add & look up to index:** If the keyword is already in the index, add the url to the list of urls associated with that keyword. If the keyword is not in the index, add an entry to the index: [keyword,[url]]

index = []

def **add\_to\_index(index,keyword,url):**

for e in index:

if(e[0] == keyword):

if url not in entry[1]: #if the url is not already present

entry[1].append(url) # add to the list of urls where the keyword is found

**return** #using break does not indicate if or not the keyword was found

**index.append([keyword,[url]])** # Code reaches here only if the keyword was not found

**return**

def **add\_page\_to\_index(index,url,content):**

newlist = content.split()

for entry in newlist:

add\_to\_index(index,entry,url)

return

def **lookup(index,keyword):**

for entry in index:

if(entry[0] == keyword):

return entry[1]

return [] #empty list

* **Split (built-in procedure):** Returns a list of words of the string “S”

**String.split(s[,sep,[maxsplit]])**

If the optional second argument “**sep**” is absent or “**None**”, the words are separated by arbitrary string of whitespce characters(space, tab, newline, formfeed, return) else it specifies the string to be used as the word separator.

If “**maxsplit**” is given, at most maxsplit number of splits occur, & the remainder of the string is return as the final element of the list (thus, the list will have at most maxsplit+1 element)

* **Web-crawler:**

def **crawl\_web**(seed):

tocrawl = [seed]

crawled = []

index = []

while tocrawl:

page = tocrawl.**pop**()

if page not in crawled:

content = **get\_page(page)**

**add\_page\_to\_index**(index,page,content)

**union**(tocrawl,**get\_all\_links(content)**)

crawled.**append**(page)

return index

print crawl\_web("http://www.udacity.com/cs101x/index.html",0)

* **Networks:** Group of entities that can communicate even though they are not all directly connected.

Network is measured in terms of two parameters:

1. **Latency:** Time its takes for the message to reach from source to destination. This is usually measure in terms of mSec
2. **Bandwidth**: amount of information (bits) that is transmitted per unit of time (sec). On the internet, this is usually in Mbps unit (million bits per sec)

Note: **traceRoute** url will list out all the intermediate hops from your location to the server location of the “url” and also provide us with the latency in mSec.

Making of network involves 3 main activities:

1. Way to encode & interpret messages: On internet, messages are encoded in bits which then can be encoded on the wire
2. Way to route messages: On internet, router figures out the next hop location on the way to destination
3. Rules to decide on who get the resource (network): there is no one rule, router decides on which packets to be sent file which is like “best effort service”

* **Protocol:** Set of rules, which are agreed upon by the entities that tells them how two entities can talk to each other. On the Internet, protocol being used most often is “HyperText Transfer Protocol (HTTP).

**Ex: Web browser (client)** send a “get” message for an “**object**” to the **web server (server)** which then replies with the “**content**” of the “object back to the client.

* **Split list:** That takes two inputs, the string to split and a string containing all of the characters considered separators. The procedure should return a list of strings that break the source string up by the characters in the splitlist.

def **split\_string(source,splitlist):**

word\_list = []

word = ''

**for char in source:** # for every character in the source

**if(char in splitlist):** # checks for every letter in splitlist

if(word): # if the word is non-empty

**word\_list.append(word)**

word = '' # empty the word to build a new word

else:

word += char #build a new word

if(word): # word after the last punctuation

word\_list.append(word)

return word\_list

* **Range function:** If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy. It generates lists containing arithmetic progressions,

**Ex:**

1. range(10) >>[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
2. range(5, 10) >> [5, 6, 7, 8, 9]
3. range(0, 10, 3) >> [0, 3, 6, 9]
4. range(-10, -100, -30) >> [-10, -40, -70]

To iterate over the indices of a sequence, combine range() and len() as follows:

a = ['Mary', 'had', 'a', 'little', 'lamb']

for i in range(len(a)):

print i, a[i]

* **Converting seconds** into hh hour(s),mm minute(s), ss second(s) format with units:

def **convert\_seconds**(seconds):

time = []

units = ['hour', 'minute', 'second']

output = ''

hh = 3600

while len(time) < 2: #untill we have calculated hrs & mins

time.append(int(seconds/hh)) # leaves seconds unchanged & provides hrs, mins in whole number

seconds = seconds % hh #denominator after division, takes decimal values also into account

hh = hh / 60

time.append(seconds) # append the remaining seconds after calculating hrs, mins

for i in range(len(time)): #i takes integer values from 0-2

if time[i] != 1:

units[i] = units[i] + 's' #plural word

for i in range(2):

output += str(time[i]) + ' ' + units[i] + ','

output += str(time[2]) + ' ' + units[2] # out side the loop to avoid the additional comma at the end

return output

* **Download calculator:** Takes as inputs a file size, the units that file size is given in, bandwidth and the units for bandwidth (excluding per second) and returns the time taken to download the file.

"<number> hours, <number> minutes, <number> seconds"

def **convert\_seconds(seconds):**

output = ''

units = ['hour','minute','second']

time = []

hh = 60\*60

while(len(time)<2):

time.append(int(seconds/hh))

seconds = seconds % hh

hh = hh/60

time.append(seconds)

for i in range(3):

if time[i] != 1:

units[i] = units[i] + 's'

output += str(time[i]) +' ' + units[i] + ', '

**return output[:-2]** #the last two characters (', ') are truncated

def **download\_time(**file\_size,d\_size,bandwidth,b\_size):

conv\_units = [['kb',2\*\*10],['kB',2\*\*10\*8],

['Mb',2\*\*20],['MB',2\*\*20\*8],

['Gb',2\*\*30],['GB',2\*\*30\*8],

['Tb',2\*\*40],['TB',2\*\*40\*8]]

**for unit,size in conv\_units**: # 1st element in sublist is unit & 2nd is size

if unit == d\_size:

file\_size = file\_size\*size

if unit == b\_size:

bandwidth = bandwidth\*size

time\_taken = **file\_size\*1.0**/bandwidth #Holds the decimal values after the integer division

return convert\_seconds(time\_taken)

# Lesson 5

* **Clock() method:** This is an in-build method which returns the current processor time as a floating point number expressed in seconds.

**import time**

def procedure():

**time.sleep(2.5)**

#measure process time

t0 = **time.clock()**

procedure()

print time.clock()-t0 # Process time in secs (0.0 sec)

#measure Wall time t0

t0 = **time.time()**

procedure()

print time.time()-t0 # Wall time in secs (2.50023 sec)

* **Eval(expression, global=none, local = none) method:** The arguments are strings and optional global & local. It evaluates the passed string as though it were Python expression and returns the result in others, it executes multiple lines of code as though they had been included instead of the single line including the eval.

**import time**

def time\_execution(code):

start = time.clock()

**result = eval(code)**

run\_time = time.clock()-start

return result, run\_time

def spin\_loop(n):

i = 0

While i < n:

i += 1

print **time\_execution(spin\_loop(10\*\*9))[1]**  >> 53.8 Secs

* **ord(‘single-character’) method:** Given a string of length one, return an integer representing the Unicode code point of the character when the argument is a Unicode object, or the value of the byte when the argument is an 8-bit string.

Ex: ord(‘a’) >> integer 97

ord(‘A’) >> integer 65

This is the inverse of ‘char()’ for 8-bit strings.

**Chr(ord(‘u’))** >> character ‘u’

* **Dictionary:** Is a mutable container type like the lists that can store any number of Python objects, including other container types. Dictionaries consist of pairs (called items) of keys and their corresponding values. Python dictionaries are also known as **associative arrays or hash tables**.

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Ex: dict1 = {‘abc’: 456}

dict1 = {‘abc’: 123, 98.6: 37}

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an **immutable data type** such as strings, numbers, or tuples.

To access dictionary elements, we can use the familiar square brackets along with the key to obtain its value.

Ex**:** dict = {‘Name: ‘Archana’, ‘ Age’ : 32, ‘Occupation’ : ‘Home maker’}

**print dict[‘Name’] >> Archana**

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error as follows:

Ex: print dict[‘Gender] **>> KeyError: Gender**

You can update a dictionary by adding a new entry or item (i.e., a key-value pair), modifying an existing entry, or deleting an existing entry as shown below in the simple example:

Ex:

1. **dict[‘Gender’] = ‘Female’**

print dict >>

{‘Name: ‘Archana’, ‘ Age’ : 32, ‘Occupation’ : ‘Home maker’, ‘Gender’: ‘Female’}

1. **del dict[‘Age’] # removes entry with key ‘Age’**

print dict >>

{‘Name: ‘Archana’, ‘Occupation’ : ‘Home maker’, ‘Gender’: ‘Female’}

1. **dict.clear() # removes all the entries from dict**

print dict >> {}

1. new\_dict ={}

new\_dict = dict.**copy()** #new\_dict is a copy of ‘dict’ dictionary

1. dict.**has\_key**(‘Married?’) #this method check for the key in the dictionary

>> False

dict.has\_key(‘Gender’)

>> True

There are two important points to remember about dictionary keys:

1. **More than one entry per key not allowed.** Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins.

Ex: dict = {‘**Name**: ‘Archana’, ‘ Age’ : 32, ‘**Name** : ‘Archu}

Print dict[‘Name’] >> Archu

Note: Value associated with a key can most certainly be a list or a dictionary itself

Ex: element = {}

**element[‘H’] = { ‘Name’: ‘Hydrogen,‘Num’ : 1,‘Weight’ : 1.00794}**

1. **Keys must be immutable.** Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'](lists which are mutable) is not allowed.

Ex: dict = {[‘**Name]**: ‘Archana’, ‘ Age’ : 32 }

Print “dict[‘Name’:]”, dict[‘Name’] >> TypeError: List objects are unhashable

**Note:** The order in which a dictionary gets printed out need not be same as what is been entered or updated by the user as display starts with the 0th buckets & goes no till n-1 bucket (like alphabetical order) content of the hash table

* **Web Crawler (Dictionary used):**

def **get\_page(url):**

**try**: #an error proof, try what is under “try” block, if none work out then execute the “except” block

**if url == "http://www.udacity.com/cs101x/index.html":**

return ('<html> <body> This is a test page for learning to crawl! '

'<p> It is a good idea to '

'<a href="http://www.udacity.com/cs101x/crawling.html">learn to '

'crawl</a> before you try to '

'<a href="http://www.udacity.com/cs101x/walking.html">walk</a> '

'or <a href="http://www.udacity.com/cs101x/flying.html">fly</a>. '

'</p> </body> </html> ')

**elif url == "http://www.udacity.com/cs101x/crawling.html":**

return ('<html> <body> I have not learned to crawl yet, but I '

'am quite good at '

'<a href="http://www.udacity.com/cs101x/kicking.html">kicking</a>.'

'</body> </html>')

**except:**

return ""

return ""

def **get\_next\_target(page):**

start\_link = page.find('<a href=')

**if (start\_link == -1):** # no links found in which case the ‘find’ method returns -1

return None,0

start\_pos = page.find('"',start\_link)

end\_pos = page.find('"',start\_pos+1)

url = page[start\_pos+1:end\_pos]

return url,end\_pos

def **get\_all\_links(page):**

links = []

while True:

url,end\_pos = get\_next\_target(page)

if url:

links.append(url)

page = page[end\_pos:]

else: # ‘get\_next\_target’ return ‘None’ when there are no more urls

break

return links

def **union(finalList,newList):**

for entry in newList:

if entry not in finalList:

finalList.append(entry)

def **crawl\_web(seed):**

tocrawl = [seed]

crawled = []

**index = {}** # Empty Dictionary

while tocrawl:

page = tocrawl.pop()

if page not in crawled:

content = get\_page(page)

add\_page\_to\_index(index,page,content)

crawled.append(page)

union(tocrawl,get\_all\_links(content))

return crawled, index

def lookup(index,keyword):

if keyword in index:

return index[keyword]

return None

**print lookup(crawl\_web("http://www.udacity.com/cs101x/index.html")[1],'good')**

Note: Crawl function returns ‘crawled’ at the 0th position and ’index’ at 1st position, as the ‘Lookup’ function only needs the ‘index’ as its 1st argument, we only extract ‘index’ from ‘crawl\_web’ function return value

* **Dictionaries of Dictionaries (of Dictionaries):**

Ex: { <hexamester>, { <class>: { <property>: <value>, ... },

... },

... }

**courses** = {

**'feb2012'**: { **'cs101'**: {'name': 'Building a Search Engine',

'teacher': 'Dave',

'assistant': 'Peter C.'},

**'cs373'**: {'name': 'Programming a Robotic Car',

'teacher': 'Sebastian',

'assistant': 'Andy'}},

**'apr2012'**: { **'cs101'**: {'name': 'Building a Search Engine',

'teacher': 'Dave',

'assistant': 'Sarah'},

**'cs212'**: {'name': 'The Design of Computer Programs',

'teacher': 'Peter N.',

'assistant': 'Andy',

'prereq': 'cs101'},

**'cs253'**:

{'name': 'Web Application Engineering - Building a Blog',

'teacher': 'Steve',

'prereq': 'cs101'},

**'cs262'**:

{'name': 'Programming Languages - Building a Web Browser',

'teacher': 'Wes',

'assistant': 'Peter C.',

'prereq': 'cs101'},

**'cs373'**: {'name': 'Programming a Robotic Car',

'teacher': 'Sebastian'},

**' cs387'**: {'name': 'Applied Cryptography',

'teacher': 'Dave'}},

**'jan2044'**: { **'cs001'**: {'name': 'Building a Quantum Holodeck',

'teacher': 'Dorina'},

**'cs003'**: {'name': 'Programming a Robotic Robotics Teacher',

'teacher': 'Jasper'},

}

}

#Returns a list of all the courses offered in the given hexamester

def **courses\_offered(courses, hexamester):**

res = []

for c in **courses[hexamester]**:

res.append(c)

return res

# Rturns True if the input course is offered in the input hexamester, and returns False otherwise

def **is\_offered(courses, course, hexamester):**

if course in courses[hexamester]:

return True

return False

# Returns a list of string representing the hexamesters when the input course is offered

def **when\_offered(courses,course):**

output = []

for hexamester in courses:

if course in courses[hexamester]:

output.append(hexamester)

return output

#Takes as input a courses structure and a person and returns a Dictionary that describes all the courses the person is involved in

def **involved(courses, person):**

output = {}

for hexamester in courses:

for course in courses[hexamester]:

for key in courses[hexamester][course]:

if courses[hexamester][course][key] == person:

if hexamester in output:

**output [hexamester].append(course**) # append to the list of course for the hexamester

else:

**output [hexamester] = [course]** # add new list of hexamester and course to the output

return output

* **Manual creation of hashtable/dictionary:**

def **make\_hashtable(size):**

table = []

for unused in range(0, size): # produces numbers from 0 to size-1

table.append([])

return table

def **hash\_string(key, bucket):**

location = 0

for letter in key:

location += ord(letter) # addes the ASCIIvalue of each of the charaters in the key

return location % bucket # provides the remainder which indicates the location of the bucket where the key would be places assuming the numbers of buckets in the hashtable is 'buckets'

def **hashtable\_get\_bucket(htable, key):**

return htable[hash\_string(key, len(htable))] #returns the hashtable entry

def **find\_key(bucket,key):**

for entry in bucket: #list of [key, value] pairs

if entry[0] == key:

return entry

return None

def **hashtable\_update(htable, key, value):**

bucket = hashtable\_get\_bucket(htable, key) # bucketlist!!

entry = find\_key(bucket,key)

if entry:

entry[1] = value

else:

bucket.append([key, value])

def **hashtable\_lookup(htable, key):**

bucket = hashtable\_get\_bucket(htable, key)

entry = find\_key(bucket,key)

if entry:

return entry[1]

return None

table = make\_hashtable(10)

hashtable\_update(table, 'Python', 'Monty')

hashtable\_update(table, 'CLU', 'Barbara Liskov')

hashtable\_update(table, 'Python', 'Guido van Rossum')

print hashtable\_lookup(table, 'Python')

# Lesson 6

ThinkPython

Important points to be noted- https://www.thenewboston.com

1. **Modulus operator(%)** which returns the remainder of the division can also be operated upon floating point variables too.
2. **Input([promt]) :** First it prints the string  you give as a parameter and then it waits for an input from the keyboard and returns the string of characters you typed

def sum(a,b):

return a+b

a = **input**("enter the 1st number: ") >> **enter the 1st number:** 10

b = **input**("enter the 2nd number: ") >> enter the 2nd number: 20

print sum(a,b) >> **30**

**Note**: This function is almost as the standard ‘scanf’ function in C language

1. **Pow(x,y) :** same as x\*\*y i.e xy
2. **Module ‘Math’:** as functions such as floor() and sqrt()

import math

math.**floor**(18.4) >> 18.0 i.e it rounds the value

math.**sqrt**(81.0) >>9.0

**sRoot = math.sqrt # user defined alias variable assigned to a function name**

**sRoot(81.0) >> 9.0**

1. To **add number to a string** either use the ‘str’ fuction to convert the number into string or put the number with the quotes

Ex: 'Archana is '+ str(32)+ ' years old'

'Archana is '+ '32'+ ' years old'

1. **When to use repr() in place of str()??**
2. **input() vs raw\_input():** raw\_input() takes exactly what the user typed and passes it back (string). input() takes the raw\_input() and performs an eval() on it as well i.e. that input() expects a **syntactically correct python expression** where raw\_input() does not

name = input("enter name: ")

>>enter name: **archana** # takes ‘archana’ as a variable name rather than a string(since there are no enclosing braces)

>>Traceback (most recent call last):

>>File "<pyshell#19>", line 1, in <module>

>> name = input("enter name: ")

>>File "<string>", line 1, in <module>

>>NameError: name 'archana' is not defined

name = input("enter name: ") >>enter name: **'archana'**

print name >>archana

**me = 'archana'**

name = input("enter name: ") >>enter name: **me**

print name >>archana

name = **raw\_input**("enter name: ") >>enter name: **archana**

print name >>archana

1. **Slicing with 1-3 parameters:** 1st parameter is the starting point, 2nd parameter is the end point and 3rd is the step to increment the start location till end.

**num = [0,1,2,3,4,5,6,7,8,9]**

1. num[-6:-1] >>[4, 5, 6, 7, 8]
2. num[-5:] >>[5, 6, 7, 8, 9]
3. num[:-5] >>[0, 1, 2, 3, 4]
4. num[:] >>[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5. num[0::2] >>[0, 2, 4, 6, 8]
6. num[-1::-2] # print the numbers backward >>[9, 7, 5, 3, 1]
7. num[4:4]=[10,11,12]

num >>[0, 1, 2, 3, 10, 11, 12, 4, 5, 6, 7, 8, 9] # numbers got added starting at 4th location

1. num[7:]=[]

num >>[0, 1, 2, 3, 10, 11, 12] # number from 7th location till end got deleted

1. num[3:]= [1,2,3,4,5,6,7,8,9]

num >>[0,1,2 ,1, 2, 3, 4, 5, 6, 7, 8, 9] # input value got updated at location starting at 3rd position

1. **Editing sequences:** Sequences are a list of items, could be strings or lists but not for dictionary
2. [1,2,3]\*3 >>[1, 2, 3, 1, 2, 3, 1, 2, 3]
3. 'arch'\*3 >>'archarcharch'
4. [3]\*4 >>[3, 3, 3, 3]
5. [[]]\*3 >>[[], [], []]
6. {'key':'Value'}\*3

>>Traceback (most recent call last):

>> File "<pyshell#51>", line 1, in <module>

>>{'key':'Value'}\*3

>>TypeError: unsupported operand type(s) for \*: 'dict' and 'int'

1. **List(),del(), min(), max(),sorted() function:**
2. list1 = **list**('archana')

print list1 >>['a', 'r', 'c', 'h', 'a', 'n', 'a']

1. **del** list1[3]

print list1 >>['a', 'r', 'c', 'a', 'n', 'a']

1. **max**(list1) >>'r'
2. **min**(list1) >>'a'
3. sorted(‘Archana’) >> ['A', 'a', 'a', 'c', 'h', 'n', 'r'] # sorted in ascending order of ASCII values
4. **Methods that work on lists:**

numbers = [5,6,9,3]

1. numbers.**append**(9)

numbers >>[5, 6, 9, 3, 9] # adds the input at the end of the list

1. numbers.**count**(9) >> 2 # counts the number of occurrence of the input
2. numbers.**index**(9) >> 2 # prints the position of the 1st occurrence of the input
3. numbers.**pop**(0) >>5 # pops out the number at the input location number

numbers >>[ 6, 9, 3, 9]

1. numbers.**remove**(9)

numbers >>[6,3,9] # removes the 1st occurrence of the input

1. numbers.**sort**()

numbers >>[3,6,9] # sorts numbers in an ascending order

1. numbers.**reverse**()

numbers >>[9,6,3] # reverses the order of item in the list

1. numbers.**insert**(2,8)

numbers >>[9,6,**8**,3] # inserts the 2nd input before 1st input location in the list

numbers.**insert**(-1,4)

numbers >>[9,6,8,**4**,3]

1. numbers.**extend**(‘12345’)

numbers.**extend**(‘abc’)

numbers >>[9,6,8,4,3,**1,2,3,4,5,a,b,c**] # converts the string to a list & then appends it to the original list

1. num1 = [1,2,3]

num2 = [4,5,6]

num1.**extend**(num2)

num1 **>>[1,2,3,4,5,6]** #Concatenates two lists, this is not possible with append() method. Append would have done **[1,2,3, [4,5,6]]**

1. **Stick variables into a string:**
2. names = ('archana', 'ananath')

string = 'Hello %s, hello %s ‘ #%s indicates that it would be replaced by user data

print string % names #% indicates that the item ‘string’ expects user data which is in ‘names’

>>Hello archana, hello ananath

1. string = "Hello %s, how are you doing??"

print string % ‘Archana’ #Data expected by ‘%’ need not always be in a value in a variable

>> Hello Archana, how are you doing??

1. **join() method for lists:** used to join the items in a list with the joining item in between each of the original list items. List itself does not change in this case.

sequence = ['hi', 'hello', 'how', 'are', 'you??']

name = ' ' #Space character

name.join(sequence) #Joins the list together with what is in ‘name’ in between each of the list items

>>'hi hello how are you??' #Each list elements separated by a space

sequence

>>['hi', 'hello', 'how', 'are', 'you??']

1. **List Methods:** Original list it self is not modified as a result of any of the below methods

sequence = 'My name is Archu!!'

sequence.**lower()**

>>'my name is archu!!'

sequence.**upper()**

>>'MY NAME IS ARCHU!!'

sequence.**capitalize()**

>>'My name is archu!!'

sequence.**replace('Archu','Archana')**

>>'My name is Archana!!'

sequence

>>'My name is Archu!!'

1. **‘is’ and ‘is not’ identity operators:** Identity operators compare the memory locations of two objects

**is –** Evaluates to true if the variables on either side of the operator **point to the same object(memory)** not if two variables have the same value and false otherwise.

x = [1,2,3]

y = [1,2,3]

x is y

>>False

a = b = [5,6,7] # both the variables are pointing to the same object

a is b

>>True

b is a

>>True

**is not** – works the opposite of ‘is’ operator

1. **Using optional/named arguments:** Python allows function arguments to have default values; if the function is called without the argument, the argument gets its default value. Furthermore, arguments can be specified in any order by using named arguments.

Ex: def info(object, spacing=10, collapse=1): #function with two optional arguments

‘spacing’ and ‘collapse’ are optional, because they have default values defined while ‘object’ is required, because it has no default value.

**info(odbchelper)** #spacing gets its default value of 10 and collapse gets its default value of 1

**info(odbchelper, 12)** # collapse gets its default value of 1

**info(odbchelper, collapse=0)** # you are naming the collapse argument explicitly and specifying its value. spacing still gets its default value of 10

**info(spacing=15, object=odbchelper)** # Even required arguments (like object, which has no default value) can be named, and named arguments can appear in any order

1. **Tuples:** Are sequences, just like lists. The only difference is that tuples can't be changed i.e., tuples are immutable and use parentheses while lists use square brackets.

They are comma-separated list of values

1. tup1 = (‘Phy’, ‘Che’, 1997, 2000)

tup1 >> ('Phy', 'Che', 1997, 2000)

1. tup2 = 1,

tup2 >> (1)

1. tup3 = “a”,”b

tup3 >>(‘a’,’b’)

1. tup4 = () or tuple()

tup4 >>() #empty tuple

1. tup5 = (2)

tup5 >>5 # not a tuple as the number is not enclosed in parenthesis

1. tup5 = (2,)

tup5 >>(5,) # single element tuple

1. tup6 = (‘archu’)

tup6 >>(‘a’,’r’,’c’,’h’,’u’) # single element tuple

You can’t modify the elements of a tuple, but you can replace one tuple with another:

tup = (‘archu’)

tup = (‘A’,) + tup[1:]

tup >> (‘A’,’r’,’c’,’h’,’u’)

**Tuple assignment** is more elegant: left side is a tuple of variables and right side is a tuple of expressions or any kind of sequence (string, list or tuple).

a,b = b, a # value exchange, all expressions on RHS are evaluated before any assignments.

username, domain = ‘archanakul@gmail.com’.split(@)

print username, domain >> (‘archanakul’, ‘gmail.com’)

Functions can only return one value, but if the value is a tuple, the effect is the same as returning multiple values.

quot, rem = divmod(7,3)

Print quot, rem >>(2,1)

Tuple values can be retrieved by indexing as we do with list and in for loops indexing

**for x in (1, 2, 3):**

print x >> 1 2 3

1. **Variable-length arguments –Tuples (converts parameters to tuple):** We may need to process a function for more arguments than you specified while defining the function. These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments. An asterisk (\*) is placed before the variable name that will hold the values of all **non-keyword variable arguments.** This tuple remains empty if no additional arguments are specified during the function call.

def printinfo (arg1, \***vartuple**):

print “Requirement argument:” + arg1

print “Variable list is:”

print vartuple

**for var in vartuple:**

print var

return

printinfo(10)

>> Requirement argument: 10

>>Variable list is:

>>**()** #empty tuple

printinfo(10, 20)

>> Requirement argument: 10

>>Variable list is:

>>**(20,)** #input parameters converted to single value tuple

printinfp(10,20,30,40)

>> Requirement argument: 10

>>Variable list is:

>>**(20,30,40)**

>> 20

>>30

>>40

1. **Variable-length arguments –Dictionary (converts parameters to dictionary):** Double asterisk(\*\*) is placed before the variable name that will hold the values of all **keyword variable arguments**, variable-length argument list.

def **test\_func(arg, \*\*dictarg):**

print "Formal arg:" + arg

for key in dictarg:

print "another keyword arg : %s:%s" % (key,dictarg[key])

test\_func("Database", name = "Archana", age = 32)

>>Formal arg:Database

>>another keyword arg : age:32

>>another keyword arg : name:Archana

def **Variable\_list\_fun(first,last,\* companies, \*\*semisters):**

print "First name: " + first

print "Last name: " + last

print "list of companies: "

print”” # extra spacing

print companies #truple can not be concatenated with string unlike a single variable

print "\n Engineering Score:" # extra spacing

for key in semisters:

print "%s: %s" % (key,semisters[key])

**Variable\_list\_fun**("archana", "kulkarni","Shipara","HCL","QuEST",styear = 70, ndyear = 70, rdyear = 75, thyear = 80)

>>First name: archana

>>Last name: kulkarni

>>list of companies:

>>('Shipara', 'HCL', 'QuEST')

>>Engineering Score:

>>styear: 70

>>thyear: 80

>>ndyear: 70

>>rdyear: 75

1. **Using \*args and \*\*kwargs when calling a function:** This special syntax can be used, not only in function definitions, but also when calling a function.

def test\_func(arg1,arg2,arg3):

print "Arg1: " , arg1

print "Arg2: " , arg2

print "Arg3: " , arg3

**arg = ("two",3)**

test\_func("Archana",**\*arg**)

>>Arg1: Archana

>>Arg2: two

>>Arg3: 3

**dictionary = {"arg3": "two","arg2": 2}**

test\_func("Archana",**\*\*dictionary**)

>>Arg1: Archana

>>Arg2: 2

>>Arg3: two

1. zip()-built-in function that takes two or more sequences and “zips” them into a list of tuples where each tuple contains one element from each sequence. If the sequences are not the same length, the result has the length of the shorter one.

first = ‘archu’

last = ‘kulk’

Zip(first,last) >>[(‘a’,’k’),(‘r’,’u’),(‘c’,’l’),(‘h’,’k’)] # missing ‘u’

If you combine zip(), for() and tuple assignment, you get a useful idiom for traversing two (or more) sequences at the same time.

def has\_match(t1,t2):

for x,y in zip(t1,t2):

if x == y: #an index i such that t1[i] == t2[i]

return True

return False

1. **enumerate()** – built-in function is used when we need to traverse the elements of a sequence and their indices.

for index,element in enumerate(‘archu’):

print index,element

>> 0 a

>> 1 r

>> 2 c

>> 3 h

>> 3 u

1. **Useful Dictionary methods:** A dictionary contains keys and values
2. dict.**key**(), dict.**val**(), dict.**items**() - By default keys and values of a dictionary cannot be stored in a list. But with the keys() and values()methods, we can store these elements in a list. With items(), we receive a list of two-element tuples. Each tuple contains, as its first element, the key. And its second element is the value.

hits = {"home" : 125, "sitemap" : 27, "about" : 43}

keys = hits.keys()

print(keys)

>> ['home', 'about', 'sitemap']

values = hits.values()

print values

>>[125, 43, 27]

items = hits.items()

print items

>>[('home', 125), ('about', 43), ('sitemap', 27)]

for key,value in hits.items():

print **‘value{} at key {}’.format(value,key)**

>>value 125 at key home

>>value 43 at key about

>>value 27 at key sitemap

1. dict.**iterkeys**(), dict.**itervalues**(), dict.**iteritems**() –These iterator are primarily useful as a means of iterating over every key/value pair in a dictionary in a for loops.

**They are more memory-efficient than the keys(), values() and items() method,** as these returns each item in succession, instead of reproducing the entire dictionary as a list in memory.

1. Update()-The method update() adds dictionary dict2's key-values pairs in to dict1. This function does not return anything.

dict1 = {"Name":'Archu',"Age":32}

dict2 = {"Sex":'Female'}

dict1.update(dict2)

dict1 >>{'Age': 32, 'Name': 'Archu', 'Sex': 'Female'}

dict2 >>{'Sex': 'Female'}

1. **String methods:**
   * **str.center(width[, fillchar])** - Return centered in a string of length width. Padding is done using the specified fillchar (default is a space).

Ex: 'archu'.center(20,'\*')

>>'\*\*\*\*\*\*\*archu\*\*\*\*\*\*\*\*'

* + **str.endswith(suffix[, start[, end]])** - Return True if the string ends with the specified suffix, otherwise return False.

Ex: 'www.gmail.com'.endswith('.com')

>>True

* + **str.expandtabs([tabsize])-** Return a copy of the string where all tab characters are replaced by one or more spaces, depending on the current column and the given tab size. Tab positions occur every tabsize characters (default is 8, giving tab positions at columns 0, 8, 16 and so on)

Ex: '12\t45\t56\t'.expandtabs()

>>'12 45 56 '

* + **str.splitlines([keepends])-** Return a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true

Ex: 'ab c\n\nde fg\rkl\r\n'.splitlines()

>>['ab c', '', 'de fg', 'kl']

'ab c\n\nde fg\rkl\r\n'.splitlines(True)

>>['ab c\n', '\n', 'de fg\r', 'kl\r\n']

* + **str.format(\*args, \*\*kwargs)-** Perform a string formatting operation. The string on which this method is called can contain literal text or replacement fields delimited by braces {}. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.

Ex: a = 1+2

"Sum of 1 and 2 = {0}".format(a)

>>'Sum of 1 and 2 = 3'

* + **str.isalnum()-**all characters in the string are alphanumeric and there is at least one character
  + **str.isalpha()-**all characters in the string are alphabetic
  + **str.isdigit()-**all characters in the string are digits
  + **str.islower()-**all cased characters in the string are lowercase
  + **str.isupper()-**all cased characters in the string are uppercase
  + **str.isspace()-**there are only whitespace characters in the string
  + **str.lstrip([chars])-** Return a copy of the string with leading characters removed. ‘chars’ argument is a string specifying the set of characters to be removed(not a prefix; rather, all combinations of its values are stripped). If omitted or ‘None’, the chars argument defaults to removing whitespace.

Ex: 'www.example.com'.lstrip('cmowz.')

>>'example.com'

* + **str.rstrip([chars])-** copy of the string with trailing characters removed.

Ex: 'mississippi'.rstrip('ipz')

>>'mississ'

* + **str.strip([chars]-** Return a copy of the string with the leading and trailing characters removed.

Ex: 'www.example.com'.strip('cmowz.')

>>'example'

* + **str.rfind(sub[, start[, end]])-** highest index in the string where substring sub is found Return -1 on failure.
  + **str.rindex(sub[, start[, end]])-** Like rfind() but raises [ValueError](https://docs.python.org/2/library/exceptions.html#exceptions.ValueError) when the substring sub is not found.
  + **str.swapcase()** - return a copy of the string with uppercase characters converted to lowercase and vice versa.
  + **str.title()** - Return a title cased version of the string where words start with an uppercase character and the remaining characters are lowercase
  + **str.zfill(width)** - Pad a numeric string ‘str’ on the left with zero digits until the given width is reached. Strings starting with a sign are handled correctly

Ex: 'archu'.zfill(7)

>>'00archu'

Important points to be noted- Basic Python Tutorial-Investary(YouTube)

1. **filter(function, sequence)** - returns a sequence consisting of those items from the sequence for which **‘function(item)’ is True**. If the sequence is a string or a tuple then the result would be of the same type else it’s always a list.

def func(num):

return (num%3 == 0 or num%5 == 0)

**filter(func,range(2,25)) # list**

>>[3, 5, 6, 9, 10, 12, 15, 18, 20, 21, 24]

**filter(func,(1,2,3,4,5,6,7,8,9,10)) # tuple**

>>>(3, 5, 6, 9, 10)

1. **map(function,sequence)** – calls ‘function(item)’ for each of the sequence’s items and returns a **list of the return values.**

def func(num):

return num\*\*3

**map(func,range(1,10))**

>>[1, 8, 27, 64, 125, 216, 343, 512, 729]

def func(num):

return (num%3 == 0 or num%5 == 0)

**map(func,range(1,10))**

>>[False, False, True, False, True, True, False, False, True] # all the return value

1. **reduce(function,sequence)** – returns a single value constructed by calling the binary function ‘function’ on the first two items of the sequence, then on the result and the next item, and so on.

def add(x,y):

return x+y

**reduce(add,range(1,100))**

>>4950

1. **List Comprehension** – List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

A list comprehension consists of **brackets** containing an **expression** followed by a **‘for’ clause, then zero or more ‘for’ or ‘if’ clauses**. The result will be a new list resulting from evaluating the expression in the context of the ‘for’ or ‘if’ clauses which follow it

square = [**x\*\*2 for x in range(10)]**

square

>>>[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

tuples = [(x,y) **for** x in (1,2,3) **for** y in (3,2,4) **if** x!=y]

tuples

>>[(1, 3), (1, 2), (1, 4), (2, 3), (2, 4), (3, 2), (3, 4)]

1. **bisect.bisect\_left (a, x, lo=0, hi=len(a))** - Locate the insertion point for x in ‘a’ to maintain sorted order. If x is already present in a, the insertion point will be **before (to the left of) any existing entries**. The returned insertion point i partitions the array a into two halves so that **all(val < x for val in a[lo:i])**for the left side and **all(val >=x for val in a[i:hi])**for the right side

seq = [3,6,9,13,13,18,21,25]

def find\_lt(array,num):

i = **bisect.bisect\_left**(array,num)

print i

if i:

return array[i-1]

return ValueError

find\_lt(seq,13)

>>3, 9

1. **bisect.bisect\_right(a, x, lo=0, hi=len(a)) or bisect.bisect(a, x, lo=0, hi=len(a)) -**

Returns an insertion point which comes after (to the right of) any existing entries of *x* in *a*.

Returned insertion point *i* partitions array *a* into two halves so that

**all(val <= x for val in a[lo:i])**for the left side and

**all(val >x for val in a[i:hi])**for the right side

seq = [3,6,9,13,13,18,21,25]

def find\_le(array,num):

i = **bisect.bisect\_right(array,num)**

print i

if i:

return array[i-1]

return ValueError

find\_le(seq,13)

>>5,13

1. **setdefault(key [,default]):** If ‘key’ is in the dictionary, returns a list of its values. If not, insert key with a value of default and return default. Default defaults to ‘None’.

def invert\_dic(d):

inverse = dict()

for key,value in d.iteritems():

**inverse.setdefault(value,[]).**append(key) **# if the key is not found then we assign an empty list as a value**

return inverse

1. **Memo:** One solution is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored for later use is called a **memo**.

cache = {} #empty dictionary

def ackermann(m, n):

if m == 0:

return n+1

if n == 0:

return ackermann(m-1, 1)

try:

**return cache[m, n]** # if the key(m,n) is found then return the corresponding value from the dictionary

except KeyError:

cache[m, n] = ackermann(m-1, ackermann(m, n-1)) # creates a new key value pair in the dictionary where the key is a tuple with m,n

return cache[m, n]

print ackermann(3, 4)

Note: **Key in a dictionary has to be a hashable value i.e. nomutable hence a list cannot be a key while a tuple could be used as a key in a dictionary.**

1. **Printing the result of a loop in a single line:**

for i in range(0,5,2):

print i, **# there is a trailing comma**

>>0 2 4

for i in range(0,5,3):

print i

>>0

>>3

1. **List slicing:**

inventory = ["sord","armor","shield","potion","apples"]

chest = ["gold","silver","iron","steel"]

inventory += chest

inventory

>>['sord', 'armor', 'shield', 'potion', **'apples', 'gold'**, 'silver', 'iron', 'steel']

inventory[4:6] = ["gold","apples"] # replace a list slice

inventory

>>['sord', 'armor', 'shield', 'potion', **'gold', 'apples',** 'silver', 'iron', 'steel']

inventory[4:6] = ["replaced"] # replace a list slice

inventory

>>['sord', 'armor', 'shield', 'potion', **'replaced',** 'silver', 'iron', 'steel']

**del** inventory[1:4] # deleting a list slice

inventory

>>[**'sord', 'replaced'**, 'silver', 'iron', 'steel']

1. **get() dictionary method:**

inventory = {'sord': 'armor', 'shield':' potion', 'silver': 'iron'}

print inventory.**get**("**sord**"," Not in the dictionary")

>>armor # return the value associated with the key

print inventory.get("**iron**"," **Not in the dictionary")**

>> **Not in the dictionary** # checks for ‘iron’ in all the dictionary keys, return the 2nd parameter if not found

print inventory.get("iron")

**>>None** # checks for ‘iron’ in all the dictionary keys, return ‘None’ (default value) as the 2nd parameter is not provided

1. **set() data type:** A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Create an empty set you have to use set()

no\_dup = set()

no\_dup

>>set([]) # empty set

basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']

fruit = set(basket)

fruit

>>set(['orange', 'pear', 'apple', 'banana']) # Collection with no duplicates

name = set('archana')

name

>>set(['a', 'h', 'c', 'r', 'n']) # Collection with no duplicates

1. Since the **tuples are non-mutable sequences**, most common use of these would be **to hold constants** which should not be changed though the course of the program.
2. **Scope of a variable:** Variables are treated as local if not otherwise declared in python unlike most other programming languages

All variables have the scope of the block, where they are declared and defined in. They can only be used after the point of their declaration.

Variables don't have to be and can't be declared in the way they are declared in programming languages like Java or C. Variables in Python are implicitly declared by defining them, i.e. the first time you assign a value to a variable, this variable is declared and has automatically the data type of the object which has to be assigned to it.

Any variable that is changed or created inside of a function is local, if it hasn't been declared as a global variable. To tell Python, that we want to use the global variable, we have to explicitly state this by using the keyword "**global**", as can be seen in the following example:

Global variable belongs to the **special frame called *‘\_\_main\_\_’***

def foo():

**global s**

print(s)

s = “Only in springs, but London is great as well!” # modifies the global variable

print(s)

s = “I am looking for a course in Paris!”

if \_\_name\_\_ == **'\_\_main\_\_':**

foo()

print(s)

1. **Reading a text file**:

**print "Reading characters from a text file!!\n"**

text\_file = **open("sample.txt","r")** # file name to open & the mode in which the file needs to be opened

print **text\_file.read**(5)

print text\_file.read(10)

**text\_file.close()**

**print "Reading a line from a text file!!\n"**

text\_file = open("sample.txt","r")

print **text\_file.readline()**

print text\_file.readline()

text\_file.close()

**print "Reading entire file from a text file!!\n"**

text\_file = open("sample.txt","r")

print **text\_file.read()**

text\_file.close()

**print "Reading line from a text file into a list!!\n"**

text\_file = open("sample.txt","r")

lines = **text\_file.readlines()**

print "All the lines of the file\n" , lines

print "File contains %s line" % **len(lines)**

for line in lines:

**print line** # prints each line in a loop

text\_file.close()

1. **Writing to a text file**:

**print "Writing to a text file!!\n"**

text\_file = **open("sample.txt","w")**

**text\_file.write**("Hello there\n")

text\_file.write("How are you there??\n")

text\_file = **open("sample.txt","r")** # open the file in read mode to read form it

print text\_file.read()

**text\_file.close()**

**print "Over-writing to a text file!!\n"**

text\_file = open("sample.txt","w")

text\_file.**write**("I am doing fine how are you??\n") **# overwrite on top of what was earlier there as the file is newly opened**

text\_file = open("sample.txt","r")

print text\_file.read()

**text\_file.close()#**without closing the file handler, the write data does not reflect

1. **Exceptional Handling**: This method is very much used in cases where the input data is being taken by the user so that we don’t prompt them with error messages which might be unnecessary and unknown to them

**try**: #Python tries to execute the code under this block

num = int(raw\_input("Enter a number to be printed:"))

**except(ValueError):** #Python executes this block wen there is ValueError

print "Incorrect value!!!\n"

except(**IOError**):

print "Incorrect IO!!!\n"

except(**SyntaxError**):

print "Incorrect Syntax!!!\n"

**except():**#Python executes this block wen there is any error basically but in this case it is executed wen there errors other than Value, IO, Syntax errors as the except blocks work as ‘elif’ blocks

print "All other errors!!\n"

**else:** # this block is executed if there are no errors

print "You entered %s as a number!!" % num

**else:**

print "You entered %s as a number!!" % num

1. **Pickles and Shelves**: This

**import cPickle,shelve**

print "Pickle List\n"

colors = ["red","blue","black","white"]

shapes = ["triangle","Square","circle","oval"]

pickle\_file = open("**pickle.dat**","w")

**cPickle.dump**(colors,pickle\_file)

cPickle.dump(shapes,pickle\_file)

print "color:\n", colors,"\n Shapes\n",shapes

pickle\_file.close()

pickle\_file = open("pickle.dat","r")

shapes = **cPickle.load**(pickle\_file)

colors = cPickle.load(pickle\_file)

print "color:\n", colors,"\n Shapes\n",shapes

pickle\_file.close()

print "Shelfing List\n"

shelves = **shelve.open("shelf.dat")**

**shelves["color"]** = ["red","blue","black","white"] #key values pairs as in dictionaries

shelves["shapes"] = ["triangle","Square","circle","oval"]

**shelves.sync()**

for shelf in shelves: # shelves work just like dictionaries

print shelf, ":", shelves[shelf]

**shelves.close()**