Estructura general de un programa en Python

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import <biblioteca>

#Declaración de funciones.

#Código principal.

if __name__ == “__main__”:
    #Instrucciones
```

Paralelo entre instrucciones en seudocódigo y lenguaje Python.

Asignación:
La asignación consiste, en el paso de valores o resultados a una zona de la memoria. Dicha zona será reconocida con el nombre de la variable que recibe el valor. La asignación se puede clasificar de la siguiente forma:

```
nombre_variable = valor
```

Tanto en seudocódigo como en Python su sintaxis es la misma, a la izquierda del signo igual debe ir el identificador (nombre) de la variable y a la derecha puede ir una expresión algebraica/aritmética, una constante, otra variable.

Ejemplos

```
a = 1
b = 3*a + 4
c = b
d = “Algun string”
```

Lectura:

La lectura consiste en recibir desde un dispositivo de entrada (el teclado) un valor. Esta operación se representa en un seudocódigo como sigue:

```
Leer a, b
```

Donde “a” y “b” son las variables que recibirán los valores.

En Python la lectura se realiza de acuerdo al tipo de datos que se desea leer.

Instruccion para lectura en Python : **`input(“Mensaje “)`**

```
numero = int(input(“Ingrese un dato: “))
```

Si es alfanumérico, se utiliza : `input(“Mensaje “)`

```
nombre = input(“Ingrese Nombre: ”)
```

Escritura:

Consiste en mandar por un dispositivo de salida (monitor o impresora) un resultado o mensaje. Este proceso se representa en un seudocódigo como sigue:

```
Escribir “El resultado es: ”, r
```

En Python la instrucción para escribir datos en pantalla es : print

```
print (“El resultado es : “, r)
```

Estructuras de Condicionales

Las estructuras condicionales comparan una variable contra otro(s) valor(es), para que en base al resultado de esta comparación, se siga un curso de acción dentro del programa. Cabe mencionar que la comparación se puede hacer contra otra variable o contra una constante, según se necesite. Existen dos tipos básicos, las simples y las múltiples.

Simples: Las estructuras condicionales simples se les conoce como “Tomas de decisión”.

```
Si <condición> entonces
    Acción(es)
Fin si
```

Dobles: Las estructuras condicionales dobles permiten elegir entre dos opciones o alternativas posibles en función del cumplimiento o no de una determinada condición.

```
Si <condición> entonces
    Acción(es)
sino
    Acción(es)
Fin si
```

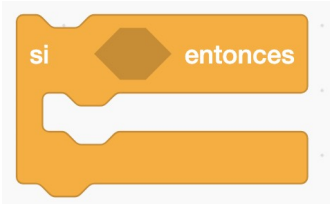
Múltiple o anidadas:

```
Si <condición> entonces
    Acción(es)
sino
    Si <condición> entonces
        Acción(es)
    si no
        Acción(es)
    Fin Si
Fin Si
```

En **Python** las estructuras condicionales recuerde el uso de la indentación, para delimitar los bloques que forman parte de las estructuras):

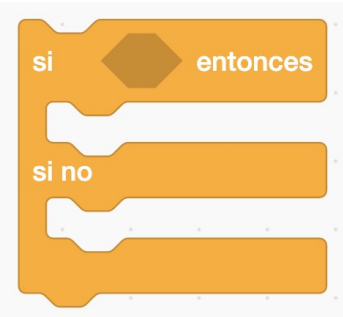
Condicionales simples.

```
if <condición>:
    Acción(es)
```



Condicionales dobles.

```
if <condición>:
    Acción(es)
else:
    Acción(es)
```



Condicionales Múltiples o anidadas

```
if <condición>:
    Acción(es)
else:
    if <condición>:
        Acción(es)
    else:
        Acción(es)
```

Otra Forma

```
if <condición>:
    Acción(es)
elif <condición>:
    Acción(es)
elif  <condición>:
    Acción(es)
else:
    Acción(es)
```

Estructura iterativa: Mientras

Esta es una estructura que repetirá un proceso durante “N” veces, donde “N” puede ser fijo o variable. Para esto, la instrucción se vale de una condición que es la que debe cumplirse para que se siga ejecutando. Cuando la condición ya no se cumple, entonces ya no se ejecuta el proceso. La forma de esta estructura es la siguiente:

```
Mientras (Condición)
    Acción_1
    Acción_2
    …
    Acción_N
Fin Mientras
```

En **Python** la estructura cíclica mientras es:

```
while (condición):
    Accion_1
    Acción_2
    …
    Acción_n
```

Estructura iterativa **PARA (For)**

El ciclo for, en Python, es aquel que nos permitirá iterar sobre una variable compleja.

```
for var in elem_iterable(lista,cadena,range,etc.):
    cuerpo del bucle
```

Definiendo funciones

En Python, la definición de funciones se realiza mediante la instrucción **def** más un nombre de función descriptivo para el cuál, aplican las mismas reglas que para el nombre de las variables- seguido de paréntesis de apertura y cierre.

Como toda estructura de control en Python, la definición de la función finaliza con dos puntos (:) y el algoritmo que la compone, irá identado con 4 espacios:

```
def mi_funcion():
    # aquí el algoritmo
```

Una función, no es ejecutada hasta tanto no sea invocada. Para invocar una función, simplemente se la llama por su nombre:

```
def mi_funcion():
    print (“Hola Mundo”)

mi_funcion()
```

Cuando una función, haga un **retorno de datos**, éstos, pueden ser asignados a una variable:

```
def funcion():
    return "Hola Mundo"

frase = funcion()
print (frase)
```

Sobre los parámetros

Un parámetro es un valor que la función espera recibir cuando sea llamada (invocada), a fin de ejecutar acciones en base al mismo. Una función puede esperar uno o más parámetros (que irán separados por una coma) o ninguno.

```
def mi_funcion(nombre, apellido):
    # algoritmo
```

Los parámetros, se indican entre los paréntesis, a modo de variables, a fin de poder utilizarlos como tales, dentro de la misma función.

Los parámetros que una función espera, serán utilizados por ésta, dentro de su algoritmo, a modo de **variables de ámbito local**. Es decir, que los parámetros serán variables locales, a las cuáles solo la función podrá acceder:

```
def mi_funcion(nombre, apellido):
    nombre_completo = nombre + " " + apellido
    print (nombre_completo)
```

Si quisiéramos acceder a esas variables locales, fuera de la función, obtendríamos un error:

```
def mi_funcion(nombre, apellido):
     nombre_completo = nombre +" "+ apellido
    print (nombre_completo)
print (nombre)
```

```
# Retornará el error: NameError: name 'nombre' is not
defined
```

Al llamar a una función, **siempre se le deben pasar sus argumentos en el mismo orden en el que los espera**.

Parámetros por omisión

En Python, también es posible, asignar valores por defecto a los parámetros de las funciones. Esto significa, que la función podrá ser llamada con menos argumentos de los que espera:

```
def saludar(nombre, mensaje='Hola'):
    print (mensaje, nombre)

saludar('Hugo Araya')
# Imprime: Hola Hugo Araya
```

```
#Programa 1
nota1 = float(input("Nota 1: “))
nota2 = float(input("Nota 2: “))
nota3 = float(input("Nota 3: “))
nota_ex = float(input("Nota examen: “))
nota_trab = float(input("Nota Trabajo: “))
promedio = (nota1+nota2+nota3)/3
final = promedio*0.55+nota_ex*0.3+nota_trab*0.15
print (“Nota Final del alumno: ", final)

#Programa 2
def pot(x,i):
    p=1
    j=1
    while j <=i:
        p = p*x
        j= j+1
    return p

def fact(i):
    f=1
    j=1
    while j<=i:
        f=f*j
        j = j+1
    return f

x=1
suma = 0
cant = 150
i = 0
while i<cant:
    termino = float(pot(x,i))/fact(i)
    suma = suma + termino
    i=i+1
print (suma)
```

```
#Programa 3
def es_vocal (x):
    if x == "a" or x == "e" or x == "i" or x == "o" or x
== "u":
        return True
    elif x == "A" or x == "E" or x == "I" or x == "O" or
x == "U":
        return True
    else:
        return False

x='a'

es = es_vocal(x)
print (es)
```

```
#Programa 3
#!/usr/bin/python
# -*- coding: utf-8 -*-
import os

def limpiaPantalla():
    if os.name == "nt":
        os.system("cls")
    elif os.name == "posix":
        os.system("clear")
    else:
        print  "<-No se pudo borrar la pantalla->"

def saludo(a):
     print (“Hola", a)

def incrementa(a,b):
     c = b+1
     d = b+2
     return c, d

if __name__ == "__main__":
    limpiaPantalla()
    saludo("Hugo")
    x, y = incrementa(1,2)
    print (x, y)
```

Métodos de Strings

Éstos son los métodos de cadena que soportan tanto las cadenas de caracteres de 8 bit como los objetos Unicode:

capitalize ()
Devuelve una copia de la cadena con el primer carácter en mayúscula.

count (*sub*[, *start*[, *end*]])
Devuelve cuántas veces aparece *sub* en la cadena S[*start*:*end*]. Los argumentos opcionales *start* y *end* se interpretan según la notación de corte.

encode ([*encoding*[,*errors*]])
Devuelve una versión codificada de la cadena. La codificación predeterminada es la codificación predeterminada de cadenas. El parámetro opcional *errors* fija el esquema de gestión de errores. Su valor predeterminado es 'strict', que indica que los errores de codificación harán saltar ValueError. Otros valores posibles son 'ignore' (ignorar) y 'replace' (reemplazar).

endswith (*suffix*[, *start*[, *end*]])
Devuelve verdadero si la cadena finaliza con el sufijo *suffix* especificado, en caso contrario falso. Si se da valor al parámetro opcional *start*, la comprobación empieza en esa posición. Si se da valor al parámetro opcional *end*, la comprobación finaliza en esa posición.

expandtabs ([*tabsize*])
Devuelve una copia de la cadena con todos los tabuladores expandidos a espacios. Si no se indica el paso de tabulación

tabsize se asume 8.

find (*sub*[*, start*[*, end*]])
Devuelve el menor índice de la cadena para el que *sub* se encuentre, de tal modo que *sub* quede contenido en el rango [*start*, *end*). Los argumentos opcionales *start* y *end* se interpretan según la notación de corte. Devuelve -1 si no se halla *sub*.

index (*sub*[*, start*[*, end*]])
Como find(), pero lanza ValueError si no se encuentra la subcadena.

isalnum ()
Devuelve verdadero si todos los caracteres de la cadena son alfanuméricos y hay al menos un carácter. En caso contrario, devuelve falso.

isalpha ()
Devuelve verdadero si todos los caracteres de la cadena son alfabéticos y hay al menos un carácter. En caso contrario, devuelve falso.

isdigit ()
Devuelve verdadero si todos los caracteres de la cadena son dígitos y hay al menos un carácter. En caso contrario, devuelve falso.

islower ()
Devuelve verdadero si todos los caracteres alfabéticos de la cadena están en minúscula y hay al menos un carácter susceptible de estar en minúsculas. En caso contrario, devuelve falso.

isspace ()
Devuelve verdadero si todos los caracteres de la cadena son espacio en blanco (lo que incluye tabuladores, espacios y retornos de carro) y hay al menos un carácter. En caso contrario, devuelve falso.

istitle ()
Devuelve verdadero la cadena tiene forma de título (anglosajón) y hay al menos un carácter. En caso contrario, devuelve falso. Se considera que una cadena tiene formato de título si todas sus palabras están en minúsculas a excepción de la primera letra de cada una, que debe ser mayúscula.

isupper ()
Devuelve verdadero si todos los caracteres alfabéticos de la cadena están en mayúscula y hay al menos un carácter susceptible de estar en mayúsculas. En caso contrario, devuelve falso.

join (*seq*)
Devuelve una cadena formada ppor la concatenación de todos los elementos de la secuencia *seq*. Los elementos se separan por la cadena que proporciona el método. Se lanza TypeError si alguno de los elementos no es una cadena.

ljust (*width*)
Devuelve la cadena justificada a la izquierda en una cadena de longitud *width*. Se rellena la cadena con espacios. Se devuelve la cadena original si *width* es menor que len(*s*).

lower ()
Devuelve una copia de la cadena convertida en minúsculas.

lstrip ()
Devuelve una copia de la cadena con el espacio inicial eliminado.

replace (*old, new*[*, maxsplit*])
Devuelve una copia de la cadena en la que se han sustituido todas las apariciones de *old* por *new*. Si se proporciona el argumento opcional *maxsplit*, sólo se sustituyen las primeras *maxsplit* apariciones.

rfind (*sub* [,*start* [,*end*]])
Devuelve el índice máximo de la cadena para el que se encuentra la subcadena *sub*, tal que *sub* está contenido en *cadena*[*start*,*end*]. Los argumentos opcionales *start* y *end* se interpretan según la notación de corte. Devuelve -1 si no se encuentra *sub*.

rindex (*sub*[*, start*[*, end*]])
Como rfind() pero lanza ValueError si no se encuentra *sub*.

rjust (*width*)
Devuelve la cadena justificada a la derecha en una cadena de longitud *width*. Se rellena la cadena con espacios. Se devuelve la cadena original si *width* es menor que len(*s*).

rstrip ()
Devuelve una copia de la cadena con el espacio al final suprimido.

split ([*sep* [*,maxsplit*]])
Devuelve una lista de las palabras de la cadena, usando *sep* como delimitador de palabras. Si se indica *maxsplit*, se devolverán como mucho *maxsplit* valores (el último elemento contendrá el resto de la cadena). Si no se especifica *sep* o es None, cualquier espacio en blanco sirve de separador.

splitlines ([*keepends*])
Devuelve una lista de las líneas de la cadena, dividiendo por límites de línea. No se incluyen los caracteres limitadores en la lista resultante salvo que se proporcione un valor verdadero en *keepends*.

startswith (*prefix*[*, start*[*, end*]])
Devuelve verdadero si la cadena comienza por *prefix*, en caso contrario, devuelve falso. Si se proporciona el parámetro opcional *start*, se comprueba la cadena que empieza en esa posición. Si se proporciona el parámetro opcional *end*, se comprueba la cadena hasta esa posición.

strip ()
Devuelve una copia de la cadena con el espacio inicial y final suprimido.

swapcase ()
Devuelve una copia de la cadena con las mayúsculas pasadas a minúsculas y viceversa.

title ()
Devuelve una versión con formato título, es decir, con todas las palabras en minúsculas excepto la primera letra, que va en mayúsculas.

translate (*table*[*, deletechars*])
Devuelve una copia de la cadena donde se han eliminado todos los caracteres de *deletechars* y se han traducido los caracteres restantes según la tabla de correspondencia especificada por la cadena *table*, que debe ser una cadena de longitud 256.

upper ()
Devuelve una copia de la cadena en mayúsculas.

TUPLAS

En Python existen varias estructuras de datos que permiten almacenar un conjunto de datos.

Una tupla es una colección de datos no necesariamente del mismo tipo que se los accede por medio de subíndices.

Definición de una tupla:

```
tupla1=('hugo', 18, 1.92)
```

Hemos definido una tupla de tres elementos.

El primer elemento es de tipo cadena de caracteres, el segundo un entero y finalmente un valor real.

Cada elemento de una tupla se los separa por una coma.
Para acceder a los elementos lo hacemos por medio del nombre de la tupla y un subíndice numérico:

```
print tupla1[0]
```

Los elementos de la tupla comienzan a numerarse a partir de cero y utilizamos los corchetes para hacer referencia al subíndice.

Si queremos en algún momento saber la cantidad de elementos de una tupla debemos llamar la función len:

```
print len(tupla1)   #imprime un 3
```

con dicha función podemos disponer una estructura repetitiva para imprimir todas las componentes de la tupla con el siguiente algoritmo:

```
tupla1=('hugo',18,1.92)
indice=0
while indice<len(tupla1):
    print tupla1[indice]
    indice=indice+1
```

Veremos en el próximo concepto que hay una estructura repetitiva que nos facilita recorrer los elementos de una tupla.

Una vez definida la tupla no se pueden modificar los valores almacenados.

La función print puede recibir como parámetro una tupla y se encarga de mostrarla en forma completa:

```
print tupla1
```

De todos modos cuando tenemos que acceder a algún elemento de la tupla debemos hacerlo mediante un subíndice entre corchetes.

La característica fundamental de una tupla es que una vez creada no podemos modificar sus elementos, ni tampoco agregar o eliminar.

Ejercicio

Definir dos tuplas que almacenen en una los nombres de los meses y en otra la cantidad de días que tiene cada mes del año. Luego mostrar el contenido almacenado en las mismas.

```
meses=('enero',   'febrero',   'marzo',   'abril',
'mayo', 'junio', 'julio', 'agosto', 'septiembre',
'octubre', 'noviembre', 'diciembre')
cantidaddias=(31, 28, 31, 30, 31, 30, 31, 31, 30,
31, 30, 31)
indice=0
while indice<len(meses):
    print meses[indice]
    print ':'
    print cantidaddias[indice]
    indice=indice+1
```

Ejercicio

Definir una tupla que almacene 5 enteros. Implementar un algoritmo que imprima la suma de todos los elementos.

```
tupla1=(6, 33, 56, 3, 45)
suma=0
indice=0
while indice<len(tupla1):
    suma=suma+tupla1[indice]
    indice=indice+1
print 'El contenido de la tupla es:' + str(tupla1)
print 'La suma es:' + str(suma)
```

Para recorrer una tupla es muy común utilizar la estructura repetitiva for.Veamos con un ejemplo la sintaxis de esta estructura repetitiva:

```
tupla1=(‘hugo’, 23, 1.92)
for elemento in tupla1:
    print elemento
```

Como podemos ver la instrucción for requiere una variable (en este ejemplo llamada elemento), luego la palabra clave in y por último la tupla.
El bloque del for se ejecuta tantas veces como elementos tenga la tupla, y en cada vuelta del for la variable elemento almacena un valor de la tupla1.

Esta estructura repetitiva se adapta mucho mejor que el while para recorrer este tipo de estructuras de datos.

Ejercicio

Definir una tupla con 5 valores enteros. Imprimir los valores mayores o iguales a 18.

```
tupla1=(45, 34, 2, 56, 1)
for elemento in tupla1:
    if elemento>=18:
        print elemento
```

Ejercicio

Definir una tupla con 10 edades de personas. Imprimir la cantidad de personas con edades superiores a 20.

```
tupla1=(45, 78, 3, 56, 3, 45, 34, 2, 56, 1)
cantidad=0
for elemento in tupla1:
    if elemento>20:
        cantidad=cantidad+1
print 'Personas con edades superiores a 20:'
print cantidad
```

El lenguaje Python nos permite rescatar una "porción" de una tupla, es decir un trozo de la misma.Si tenemos la siguiente tupla:

```
tupla1=(1, 7, 20, 40, 51, 3)
tupla2=tupla1[0:4]
print tupla2
```

El resultado es una tupla con cuatro valores:

```
(1, 7, 20, 40)
```

Es decir indicamos como subíndice un rango de valores, en este caso desde la posición 0 hasta la posición 4 sin incluirla.

Podemos no indicar alguno de los dos rangos:

```
tupla1=(1, 7, 20, 40, 51, 3)
tupla2=tupla1[3:]
print tupla2
```

El resultado es una tupla con tres valores, desde la posición 3 hasta el final de la tupla:

```
(40, 51, 3)
```

En caso de no indicar el primer rango:

```
tupla1=(1, 7, 20, 40, 51, 3)
tupla2=tupla1[:2]
print tupla2
```

El resultado es una tupla con dos valores, desde el principio de la tupla hasta la posición 2 sin incluirla:

```
(1, 7)
```

Ejercicio

Definir una tupla con los nombres de los meses. Generar dos tuplas que almacenen los primeros 6 meses la primera y los siguientes 6 meses la segunda.

```
meses=('enero',   'febrero',   'marzo',   'abril',
'mayo', 'junio', 'julio', 'agosto', 'septiembre',
'octubre', 'noviembre', 'diciembre')
tupla1=meses[:6]
tupla2=meses[6:]
print tupla1
print tupla2
```

Ejercicio

Almacenar en una tupla 5 nombres. Luego generar un valor aleatorio entre 2 y 4. Copiar a una tupla el nombre de la posición indicada por el valor aleatorio y los nombres que se encuentran en la posición anterior y posterior.

```
import random

nombre=('juan', 'ana', 'luis', 'carlos', 'ramon')
ale=random.randint(1, 3)
tresnombres=nombre[ale-1:ale+2]
print tresnombres
```

CADENAS DE CARACTERES

Una cadena de caracteres permite almacenar un conjunto de caracteres. Su funcionamiento es similar a una tupla.Para inicializar un string utilizamos el operador de asignación.

```
nombre='Juan Pablo'
```

Podemos utilizar las comillas simples o dobles para su inicialización:

```
mail='jose@gmail.com'
```
o
```
mail="jose@gmail.com"
```

Para conocer el largo de un string podemos utilizar la función len:

```
print len(mail)
```

Para acceder a un caracter particular del string lo hacemos indicando un subíndice entre corchetes:

```
print mail[0]  #Imprimimos el primer caracter
```

El lenguaje Python nos permite rescatar una "porción" de un string con la misma sintaxis que trabajamos las tuplas:

```
nombre='Jose Maria'
print nombre[1:4]  #ose
print nombre[:4] #Jose
print nombre[5:] #Maria
```

Los string son inmutables, es decir que no podemos

modificar su contenido luego de ser inicializados:

```
titulo='Administracion'
titulo[0]='X' # Esto produce un error
```

Esto no significa que no podemos utilizar la variable para que referencie a otro string:

```
nombre='Jose'
print nombre
nombre='Ana'
print nombre
```

Para concatenar string Python permite utilizar el operador +.Si tenemos tres string y queremos almacenar sus contenidos en un cuarto string podemos codificarlo de la siguiente manera:

```
cadena1='uno'
cadena2='dos'
cadena3='tres'
total=cadena1+cadena2+cadena3
print total #unodostres
```

También Python define el operador * para los string. El resultado de multiplicar un string por un entero es otro string que repite el string original tantas veces como indica el número.

#si queremos un string con 80 caracteres de subrayado, la forma más sencilla es utilizar la siguiente expresión:

```
separador='_'*80
print separador
```

Los operadores relacionales definidos para los string son:

> Mayor
>= Mayor o igual
< Menor
<= Menor o igual
== Igual
!= Distinto

Si queremos saber si un string es mayor alfabéticamente que otro utilizamos el operador >

```
nombre1='CARLOS'
nombre2='ANABEL'
if nombre1>nombre2:
    print nombre1+' es mayor alfabéticamente que '+nombre2
```

si queremos saber si dos variables tienes en mismo contenido:

```
nombre1='CARLOS'
nombre2='CARLOS'
if nombre1==nombre2:
   print 'Las dos variables tienen el mismo contenido:'+nombre1
```

Ejercicio

Elaborar una función que reciba un string y retorne la cantidad de vocales que tiene.

```
def cantidadvocales(cadena):
    cant=0
```

```
    for letra in cadena:
        if letra=='a':
            cant=cant+1
        if letra=='e':
            cant=cant+1
        if letra=='i':
            cant=cant+1
        if letra=='o':
            cant=cant+1
        if letra=='u':
            cant=cant+1
        if letra=='A':
            cant=cant+1
        if letra=='E':
            cant=cant+1
        if letra=='I':
            cant=cant+1
        if letra=='O':
            cant=cant+1
        if letra=='U':
            cant=cant+1
    return cant

print 'Cantidad de vocales en la palabra Hola:'
print cantidadvocales('Hola')
print 'Cantidad de vocales en la palabra Computadora:'
print cantidadvocales('Computadora')
```

Problema Propuesto

Elaborar las siguientes funciones:
- Una función que reciba un string y nos retorne el primer caracter.

- Una función que reciba un apellido y un nombre, y nos retorne un único string con el apellido y nombre concatenados y separados por una coma.
- Una función que reciba dos string y nos retorne el que tiene menos caracteres.

```
def primercaracter(cadena):
    return cadena[0]

def concatenar(apellido,nombre):
    return apellido + ',' + nombre

def menor(cadena1,cadena2):
    if len(cadena1)<len(cadena2):
        return cadena1
    else:
        return cadena2

cad='Hola Mundo'
print 'Primer caracter de ' + cad + ' es ' +
primercaracter(cad)
nom='juan'
ape='rodriguez'
print 'Apellido y nombre concatenados:' +
concatenar(ape, nom)
cad1='Hola'
cad2='Fin'
print 'De: ' + cad1 + ' y ' + cad2 + ' tiene menos
caracteres ' + menor(cad1, cad2)
```

LISTAS

Una lista es una colección de datos no necesariamente del mismo tipo que se los accede por medio de subíndices. La diferencia fundamental de una lista con una tupla es que podemos modificar la estructura luego de haberla creado.

Hay varias formas de crear una lista, la primera y más sencilla es enumerar sus elementos entre corchetes y separados por coma:

```
lista1=['juan', 'ana', 'luis']
```

Para acceder a sus elementos lo hacemos indicando un subíndice subíndice:

```
print lista1[0]
```

Como decíamos la diferencia con una tupla (son inmutables) es que podemos modificar la lista luego de creada:

```
lista1=[10, 15, 20]
print lista1
lista1[0]=700 # modificamos el valor almacenado en la primer componente de la lista.
print lista1
```

De forma similar a las tuplas y string la función len nos informa de la cantidad de elementos que contiene la lista:

```
lista1=[10, 15, 20]
print len(lista1) # imprime un 3
```

Es muy común emplear la estructura for in para recorrer y

rescatar cada elemento de la lista, la variable elemento almacena en cada ciclo del for un elemento de la lista1, comenzando por el primer valor:

```
lista1=['juan', 23, 1.92]
for elemento in lista1:
    print elemento
```

Si queremos saber si un valor se encuentra en una lista existe un operador llamado in:

```
lista1=[12, 45, 1, 2, 5, 4, 3, 55]
if 1 in lista1:
    print 'El valor 1 está en la lista '
else:
    print 'El valor 1 no está en la lista '
print lista1
```

Python define los operadores + y * para listas, el primero genera otra lista con la suma de elementos de la primer y segunda lista. El operador * genera una lista que repite tantas veces los elementos de la lista como indica el valor entero seguido al operador *.

```
lista1=[2, 4, 6, 8]
lista2=[10, 12, 14, 16]
listatotal=lista1+lista2
print listatotal
```

Luego empleando el operador *:

```
lista1=['juan', 'carlos']
producto=lista1*3
print producto
```

El resultado de este algoritmo es:

```
['juan', 'carlos', 'juan', 'carlos', 'juan', 'carlos']
```

También con listas podemos utilizar el concepto de porciones que nos brinda el lenguaje Python:

```
lista=[2, 4, 6, 8, 10]
print lista[2, 4]   #[6, 8]
print lista[:3]    #[2, 4, 6]
print lista[3:]    #[8, 10]
```

Como se vio podemos modificar el contenido de elementos de la lista asignándole otro valor:

```
lista=[2, 4, 6]
lista[1]=10
print lista  #[2, 10, 6]
```

Podemos borrar elementos de la lista utilizando la función del:

```
lista=[2, 4, 6]
del(lista[1])
print lista  #[2, 6]
```

Además podemos utilizar porciones para borrar un conjunto de elementos de la lista:

```
lista=[2, 4, 6]
del(lista[1:])
print lista  #[2]
```

También podemos añadir elementos a una lista:

```
lista=[5, 10, 11, 12]
lista[1:1]=[6, 7, 8, 9]
print lista #[5, 6, 7, 8, 9, 10, 11, 12]
```

Ejercicio

Definir una lista con edades de personas, luego borrar todos los elementos que sean menores a 18.

```
edades=[23, 5, 67, 21, 12, 4, 34]
indice=0
while indice<len(edades):
    if edades[indice]<18:
        del(edades[indice])
    else:
        indice=indice+1
print edades
```

Problema Propuesto

Definir una lista con un conjunto de nombres, imprimir la cantidad de comienzan con la letra a:

```
nombres=['ariel',   'marcos',   'ana',   'luis',
'pedro', 'andres']
cant=0
for nom in nombres:
    if nom[0]=='a':
        cant=cant+1
print nombres
print 'Cantidad de nombres que comienzan con a
es:'
print cant
```

INDICES NEGATIVOS

Hemos visto que en Python accedemos a los elementos de una lista, tupla y string mediante un subíndice que comienza a numerarse a partir de cero:

```
tupla=(2, 4, 6, 8, 10)
print tupla[0]   # 2
lista=[2, 4, 6, 8, 10]
print lista[0]   # 2
cadena='hola'
print cadena[0]   # h
```

Si queremos acceder a la última componente podemos hacerlo:

```
tupla=(2, 4, 6, 8, 10)
print tupla[len(tupla)-1] # 10
```

Pero Pyton tiene integrado en el lenguaje el acceso de los elementos de la secuencia mediante índices negativos, por ejemplo si queremos acceder a la última componente luego podemos hacerlo con la siguiente sintaxis:

```
tupla=(2, 4, 6, 8, 10)
print tupla[-1]   # 10
lista=[2, 4, 6, 8, 10]
print lista[-1]   # 10
cadena='hola'
print cadena[-1]   # a
```

Es mucho más cómodo utilizar esta segunda forma para acceder a los elementos de una lista, tupla o cadena de caracteres.

Si queremos imprimir los elementos de una tupla en forma inversa (es decir desde el último elemento hasta el primero) podemos hacerlo con el siguiente algoritmo:

```
tupla=(2, 4, 6, 8, 10)
indice=-1
for x in range(0, len(tupla)):
    print tupla[indice] # 10 8 6 4 2
    indice=indice-1
```

Ejercicio

Inicializar una variable con un valor aleatorio comprendido entre 1 y 1000000, verificar si es capicúa, es decir si se lee igual de izquierda a derecha como de derecha a izquierda.

```
print valor
cadena=str(valor)
indice=-1
iguales=0
for x in range(0, len(cadena)/2):
    if cadena[x]==cadena[indice]:
        iguales=iguales+1
    indice=indice-1
if iguales==(len(cadena)/2):
    print 'Es capicua'
else:
    print 'No es capicua'
```

Problema Propuesto

Definir una lista con una serie de elementos. Intercambiar la información del primero con el último de la lista.

```
lista=['juan', 'ana', 'luis', 'pedro']
print lista
aux=lista[0]
lista[0]=lista[-1]
lista[-1]=aux
print lista
```

DICCIONARIOS

Las estructuras de datos vistas hasta ahora (tuplas y listas) utilizan un entero como subíndice. La estructura de datos tipo diccionario utiliza una clave para acceder a un valor. El subíndice puede ser un entero, un string etc.

Un diccionario vincula una clave y un valor.

Si queremos almacenar la cantidad de productos que tenemos en stock podemos implementar un diccionario donde utilizamos como clave el nombre del producto y como valor la cantidad de productos disponibles.

Podemos definir un diccionario con la siguiente sintaxis:

```
productos={'manzanas':23, 'peras':50, 'papas':120}
```

Luego si queremos acceder a la cantidad de productos en stock utilizamos como subíndice el nombre del producto:

```
print productos['manzanas']
```

La línea anterior produce como resultado el valor 23.
Mediante un diccionario asociamos para toda clave un valor.
Podemos borrar cualquier entrada dentro del diccionario empleando la siguiente sintaxis:

```
del(productos['manzana'])
```

Luego si imprimimos el diccionario en forma completa tenemos como resultado que ya no contiene la entrada 'manzana':

```
print propductos
```

```
{'peras': 50, 'papas': 120}
```

Podemos modificar el valor asociado a una clave mediante una simple asignación:
productos['papas']=5

Luego si imprimimos el diccionario obtenemos como resultado:

```
{'peras': 50, 'papas': 5}
```

Podemos conocer en cualquier momento la cantidad de pares clave-valor que contiene nuestro diccionario mediante la función len:

```
print len(productos)
```

Problema resuelto

Crear un diccionario cuyas claves sean palabras en inglés y su valor almacena la palabra traducida al castellano.

```
diccionario={'house':'casa','red':'rojo','bed':'cama','window':'ventana'}
print diccionario['house']
print diccionario['red']
```

Problema Propuesto

Crear un diccionario asociando nombres de países y cantidades de habitantes. Imprimir luego el diccionario.

```
paises={‘chile’:18000000,    'argentina':40000000, 'españa':46000000, 'brasil':190000000}
print paises
```

FORMATEO DE STRING

Cuando tenemos que combinar cadenas de caracteres e incorporarle valores almacenados en otras variables el lenguaje Python nos suministra una técnica para incrustar valores dentro de la cadena.

Veamos mediante ejemplos cual es la sintaxis para plantear estos formatos de cadenas:

```
x1=10
x2=5
x3=20

print 'El contenido de la primer variable es %d, de la segunda %d y la tercera %d' % (x1,x2,x3)
```

Si bien podemos utilizar el operador + para ir concatenando cadenas y variables esta técnica hace que nuestro programa sea más legible sobre todo cuando tenemos que sustituir varias variables dentro de una cadena.

Primero indicamos entre comillas la cadena y utilizamos el símbolo % para indicar el lugar donde se sustituirá el valor, debemos indicar luego del caracter % el tipo de dato (en nuestro ejemplo un valor decimal (d))

Luego de la cadena debemos indicar una tupla con los valores o variables de donde se obtendrán los datos. Entre la cadena de formato y la tupla debemos disponer el caracter % (aquí el caracter porcentaje tiene un objetivo distinto que dentro de la cadena de formato)

Se especifican distintos caracteres para cada tipo de dato a sustituir:

```
x=10
g=10.2
cadena='juan'

print 'El valor entero %d el valor real %f y la cadena %s' % (x,g,cadena)
```

En el ejemplo propuesto vemos que podemos utilizar los caracteres de formato d (decimal), f (float) y s (string).

Es importante el orden de los valores de la tupla ya que el lenguaje procesa los datos en forma secuencia, es decir cada vez que debe sustituir un valor en la cadena extrae de la tupla el siguiente valor.

Cuando damos formato a una variable real (con coma) podemos disponer dos valores previos al caracter f:

```
g=20.5498
print 'El valor real es %10.2f' % (g)
```

El primer valor indica el largo total a reservar y el segundo la cantidad de decimales.
Podemos convertir el valor decimal a tipo octal o hexadecimal:

```
x=255
print 'Decimal %d en hexadecimal es %x y en octal %o' % (x,x,x)

 #Decimal 255 en hexadecimal es ff y en octal 377
```

No es obligatorio que el formato de cadena se utilice siempre en un print, podemos almacenar el resultado del formato en otra variable string (que podemos eventualmente almacenarla en una base de datos por ejemplo).

```
vx=10
vy=90
resultado='(%d, %d)' % (vx, vy)
print resultado # (10,90)
```

También podemos indicar un valor entero en el formato para los tipos de datos enteros y string:

```
x1=100
x2=1500
x3=5
print '%5d' % (x1)
```

```
print '%5d' % (x2)
print '%5d' % (x3)
```

El resultado por pantalla es:

```
  100
 1500
    5
```

Es decir reserva en este caso 5 espacios para el entero y hace la alineación a derecha.

Si indicamos un valor negativo los datos se alinean a izquierda:

```
animales=['perro', 'elefante', 'pez']
for elemento in animales:
    print '%20s' % elemento
for elemento in animales:
    print '%-20s' % elemento
```

El resultado de ejecutar el programa es:

```
               perro
            elefante
                 pez
perro
elefante
pez
```

Problema resuelto

Imprimir todos los números desde el 1 hasta el 255 en decimal, octal y hexadecimal.

```
for num in range(1,256):
```

```
    print '%3d   %3o   %3x' % (num, num, num)
```

Problema Propuesto

Almacenar en una lista los nombres de personas y en otra los sueldos que cobran cada uno. Hacer que para el índice cero de cada componente representen los datos de una persona y así sucesivamente.

Imprimir un nombre por línea de la pantalla y hacer que los sueldos aparezcan correctamente alineados las columnas de unidades, decenas, centenas etc.

```
nombres=['juan', 'ana', 'luis']
sueldos=[1500.55, 2700.00, 910.66]
for indice in range(0,len(nombres)):
    print '%-20s  %10.2f' % (nombres[indice],
sueldos[indice])
```