1. Python—fundamentos parte 1:

Características básicas:

Python es un lenguaje de programación imperativo, modular, estructurado y orientado a objetos, pero no obliga a los programadores a hacer las cosas pensando en el paradigma orientado a objetos. Esto significa que en Python no hay una relación uno-a-uno entre los archivos de código fuente y las clases. En Java cada archivo (con extensión `.java`) corresponde a (contiene) la definición de una clase (y al compilar el código fuente se produce un archivo equivalente con extensión `.class`). El programa principal mismo debe tener un método `main` definido dentro de una clase para que se pueda ejecutar después de compilarlo. En Python, por otra parte, los programas fuente (que típicamente tienen la extensión `.py`) pueden no contener la definición de ninguna clase (si el programador no desea hacer uso explícitamente de las características orientadas a objetos del lenguaje), o puede tener la definición de una o varias clases. Las clases pueden estar anidadas (cada clase interna sólo se puede usar—instanciar—dentro de la clase externa en la que está definida) o pueden ser independientes. No hay restricciones acerca de estos aspectos en Python. No es necesario tener un método `main` (o equivalente) en ninguna parte tampoco. El código fuente en Python es leído y ejecutado de arriba a abajo y puede incluir definiciones de funciones y/o clases así como instrucciones aisladas (incluyendo invocaciones a funciones) en cualquier combinación. Lo que sí puede ser importante (depende de cómo esté configurado el Python que se esté utilizando) es el orden de las definiciones: en algunos casos para que una instrucción pueda invocar a una función no-predefinida, la definición de dicha función tiene que haber aparecido previamente a la instrucción que la invoca dentro del código fuente del programa en Python.

En Python no se tiene que usar `;` para terminar cada línea/instrucción dentro del código y se pueden imprimir valores en la consola usando la siguiente sintaxis:

```
print("hola mundo")
```

¡En Python no se tienen que declarar los tipos de los identificadores (constantes, variables, funciones/métodos)! Simplemente se hace uso de los identificadores que se necesiten conforme se necesiten. Esto no implica que el lenguaje no entienda o no tenga el concepto de tipo, pero el lenguaje infiere los tipos de los identificadores solito ("type inference") basándose en la forma en la que se utilizan dentro de las instrucciones de un programa. En Python el tipo de los identificadores puede cambiar dinámicamente durante la ejecución del programa. Lo que sí es necesario es asegurarse de que se hayan asignado valores a los identificadores si se van a utilizar dichos identificadores dentro de expresiones que requieran que ya tengan valores. Por ejemplo,

```
print(a)
```

no es una instrucción válida si en ninguna instrucción anterior se le ha asignado un valor a la variable `a`, pero sí es válido hacer

```
a=3
print(a)
```

Las dos filosofías principales de Python son darle flexibilidad a los programadores y darle simplicidad/legibilidad a los programas.

Comentarios preliminares acerca de cómo escribir programas en Python:
Los comentarios de una sola línea en Python comienzan con `#` (en lugar de `//` como se hace en Java). Los comentarios de múltiples líneas en Python comienzan y terminan con `"""` (tres comillas dobles seguidas) en lugar de empezar con `/*` y terminar con `*/` como se hace en Java)…o con `'''` (tres comillas sencillas seguidas), pues en Python las comillas sencillas y las dobles son equivalentes (lo cual principalmente es útil en el manejo de cadenas de caracteres, pero es válido también con respecto a los comentarios).

No es obligatorio, pero la comunidad de programadores en Python por convención ha adoptado el uso de "notación serpiente" en lugar de "notación camello" (que es lo que se acostumbra en Java) para los nombres simbólicos, por ejemplo:

```
tarjeta_de_credito
```

en lugar de

```
tarjetaDeCredito
```

Las reglas que existen en Python acerca de todos los nombres simbólicos es que deben comenzar con una letra (del inglés) o guion bajo y el resto del nombre puede consistir de cualquier combinación de cero o más letras (del inglés), guiones bajos, o dígitos.

Python es un lenguaje interpretado, lo cual significa que la traducción a lenguaje de máquina (0s y 1s) y la ejecución se realizan de forma intercalada. No hay dos fases, como en los lenguajes compilados (incluyendo Java), en los que primero el compilador traduce el código fuente a lenguaje de máquina y luego se tiene que invocar otro programa que se encargue de ejecutar el programa en lenguaje de máquina (aunque en Java, si se usa un IDE como NetBeans para desarrollar los programas, el mismo IDE se encarga de hacer las dos fases, una tras otra, así es que este aspecto se vuelve transparente para el programador). Esto implica que el programador puede teclear instrucciones directamente en la consola de Python y éstas se ejecutan en ese momento, directamente, sin la necesidad de que estén almacenadas en un archivo de código fuente. Por otra parte, para desarrollar cualquier programa de complejidad no-trivial muchas veces es necesario hacer pruebas de partes del código para depurarlas poco a poco, y esto se facilita si el código fuente se almacena en un archivo (que se pueda editar, parcialmente, fácilmente y tantas veces como sea necesario hasta tener la versión final), aparte de la ventaja adicional de almacenar el programa permanentemente una vez que esté escrito. Afortunadamente, la mayoría de los ambientes de desarrollo de programas en Python también permiten esta forma de trabajar, pues tienen una consola para ejecutar código pero además una ventana para la edición de código.

En Java se agrupan líneas de código (es decir, se crean bloques) mediante el uso de llaves `{}`. En Python las sangrías son las que definen los bloques de código. Cada nivel adicional de sangría requiere agregar cuatro espacios (o un tabulador) a la izquierda de las instrucciones de ese nivel en comparación con el nivel anterior. No pueden ser ni más, ni menos, de cuatro. Python es muy

estricto con esto para darle legibilidad a los programas, pues el inicio (el lado izquierdo) de cada línea de código que pertenece a un bloque a fuerzas queda alineado con el inicio de todas las demás líneas de código pertenecientes al mismo bloque. En Python el encabezado que comienza la definición de un bloque debe ser una línea de código que termine en `:`.

En Python las funciones son objetos. Esto implica que pueden ser utilizados como datos…pero son datos especiales que también pueden ser ejecutados. La definición de una función en Python comienza por la palabra reservada `def`. El cuerpo de la función debe escribirse con un nivel adicional de anidamiento (sangría). Python sabe que termina la definición del cuerpo de la función en cuanto se regresa al nivel de sangría/anidamiento anterior. Un ejemplo de una función `a` utilizada como dato ocurre cuando se envía su nombre como argumento al invocar otra función `b`. La función `b` recibe la función `a` dentro de un parámetro (llamémoslo `c`) pero ese parámetro, ese argumento, esa función `a` (llamada `c` dentro del contexto de `b`) se puede ejecutar poniendo paréntesis junto a su nombre (que es la sintaxis que se usa para invocar a una función en Python, igual que en Java) dentro del cuerpo de `b`. Concretamente:

```
#La función 'a' imprime "Hola" en la consola:
def a():
    print("Hola")
#La función 'b' invoca a la función que recibe como parámetro:
def b(c):
    #Invoca a la función 'c' (el parámetro):
    c()
#Invoca a la función 'b', enviándole la función 'a' como parámetro.
#El resultado final debería ser que se imprima "Hola" en la consola, que es
#lo que hace 'a':
b(a)
```

En Python se pueden realizar múltiples asignaciones escribiendo una sola línea de código. Esto permite hacer un intercambio de valores ("swap") en una sola línea/instrucción, sin utilizar una variable intermedia, como se haría tradicionalmente. Debido a esto, para intercambiar los valores de `a` y `b`, en Python se puede hacer:

```
a=3
b=4
temp=a
a=b
b=temp
```

Pero también se puede lograr el mismo resultado haciendo:

```
a=3
b=4
a,b=b,a
```

Como se puede ver, el operador de asignación en Python es el mismo que en Java (`=`). Un aspecto de Python relacionado con el ejemplo anterior es que las funciones pueden regresar múltiples valores (separándolos por comas en la instrucción `return`—la cual sigue existiendo en Python al igual que en Java—dentro de la definición de la función, y recibiendo el resultado de

ejecutar la función en múltiples variables (también separadas por comas) del lado izquierdo de alguna instrucción de asignación que, de su lado derecho, invoque la función.

Operadores en Python:
En Python no existen el operador incremento (`++`) ni el operador decremento (`--`). Si uno quiere hacer lo que en C, C++ y Java se puede lograr a través de una operación como `++i;` o `i++;`, en Python se tiene que hacer explícitamente (es decir, se tiene que hacer `i=i+1`). Por otra parte, sí existen los operadores `+=`, `-=`, etc.

En Python los operadores aritméticos son muy parecidos a los de Java, pero el funcionamiento de `/` es un poco diferente. Java interpreta el operador `/` como un cociente si sus dos operandos son enteros, o como una división exacta (arrojando un resultado con parte decimal) si cualquiera de sus operandos es real (o si los dos lo son). En Python, como no se declaran los tipos de los identificadores, se necesita usar el operador `//` si se desea realizar la división entera (dando un cociente entero como resultado), pues el operador `/` siempre se interpreta como una división exacta y arroja un valor real como resultado (esta suposición le permite a Python realizar la inferencia de tipos que siempre hace "tras bambalinas"). Python tiene un operador aritmético adicional (aparte de `+`, `-`, `*`, `/`, `//` y `%`): el operador potencia (escrito `**`). En Python los operadores lógicos se escriben en inglés (`and`, `or` y `not`), en lugar de tener símbolos especiales como en Java (donde se escriben `&&`, `||` y `!`, respectivamente). En Python los operadores relacionales son los mismos que en Java (`<`, `>`, `<=`, `>=`, `==` y `!=`). Un operador adicional que existe en Python es el operador `in` de membresía en un conjunto (o en una colección en general—las colecciones se verán más adelante), que según el contexto puede usarse tanto para consultar si un elemento pertenece a una colección o no, o para extraer, uno por uno, los elementos de una colección.

Tipos de datos básicos predefinidos y conversiones entre ellos en Python:
Como se mencionó previamente, en Python no se tienen que declarar los tipos de los identificadores, lo cual no significa que Python no tiene el concepto de tipo y no realiza un análisis de los tipos de los identificadores—todo lo contrario. Se puede consultar el tipo de un identificador en cualquier momento en Python utilizando la función predefinida `type`. En Python se puede modificar el tipo de un identificador dinámicamente durante la ejecución de un programa. Algunos de los tipos predefinidos en Python son: `int` (y algunas variantes), `float`, `complex`, `bool` (con dos valores, `False` y `True`, compatibles con los valores numéricos 0 y todos los demás, respectivamente—es decir, se pueden usar valores numéricos o expresiones numéricas en las condiciones de los `if` y los `while`), `str` (y otras colecciones como `list`), `function` (por eso las funciones se pueden manipular como si fueran datos) y `type` (el tipo `type` se le asigna a los tipos de valores no-predefinidos, como cuando uno define una clase no-predefinida), así como otros tipos de asociaciones y agrupaciones de datos (como diccionarios, tuplas y conjuntos, los cuales se verán más adelante). Se pueden realizar conversiones, ya sea implícitas (también llamadas coerciones) o explícitas (también llamadas "casts"), entre algunos de estos tipos de datos. La sintaxis de una conversión explícita es "al revés" de la de Java: en Python en lugar de poner el tipo al que se quiere convertir entre paréntesis, se pone el valor que se desea convertir entre paréntesis (ver segundo conjunto de instrucciones de ejemplo dentro de este punto, abajo). Un ejemplo de una conversión implícita es:

```
a=5
b=5.5
c=a+b
print(c)
```

En este ejemplo Python automáticamente convierte el entero `a` en un número equivalente de punto flotante (con un valor de 5.0) para poder sumarlo al otro operando (que ya está expresado como un número de punto flotante desde el principio). Esta "promoción" automática de tipos (coerción) la hace Python sólo para ciertas combinaciones de tipos. Por otra parte, los siguientes son ejemplos de conversiones explícitas de tipos válidas:

```
a=int(4.5)
b=float(7)
print(a)  #Imprime 4 (truncó la parte decimal para hacer la conversión)
print(b)  #Imprime 7.0 (le insertó una parte decimal, aunque esté "vacía")
```

Otro tipo que existe de forma predefinida en Python es el tipo `None`, que tiene un solo valor posible (también escrito `None`), que es equivalente al concepto de `null` en Java (es decir, la ausencia de un objeto real en memoria como valor de cierta variable, que en Java tiene que ser de un tipo no-primitivo—concepto que no existe en Python, pues todo valor internamente se representa como si fuera un objeto). Las funciones en Python que no regresan valores (explícitamente) a través de instrucciones `return` en realidad automáticamente (e implícitamente) regresan el valor `None` (que es el único valor que existe dentro del tipo `None`), que es análogo a lo que ocurre con las funciones de tipo `void` en Java, que regresan el valor `null`.

Programación orientada a objetos en Python:
Python es un lenguaje orientado a objetos, pero es menos estricto en su forma de manejar la orientación a objetos que Java. En Java una clase puede heredar sólo de una otra clase (esto pareciera ser una limitante bastante estricta, pero adicionalmente una clase puede implementar cualquier cantidad de interfaces, lo cual hace que el concepto de herencia sea más flexible). Por otra parte, en Python se permite la herencia múltiple (y por lo tanto no existe el concepto de interfaz en el sentido orientado a objetos que usa Java para hacer más flexible la herencia). En Java se tienen cuatro niveles de visibilidad (`public`, `private`, `protected` y la cuarta categoría, que se define sin palabras reservadas pero a veces se describe como "package protected"). En Python no existen distintos niveles de visibilidad. Se tiene acceso global (público), desde cualquier parte del código, a cualquier miembro de cualquier clase (con tal de hacerlo usando el operador punto para indicar el objeto cuyo atributo se desea consultar o al cual se desea aplicar un método, usándolo igual que en Java: a la izquierda del punto se pone el nombre del objeto y a la derecha el nombre del atributo o método). En Python, debido a que no se tienen que declarar las variables antes de usarlas, tampoco se necesitan declarar los atributos de las clases antes de usarlos. Simplemente llegan a existir si se utilizan dentro de los métodos (especialmente en el método constructor) de las clases. En Python existen nombres especiales para el método constructor y algunos otros métodos frecuentemente necesitados, con funcionalidades genéricas, dentro de las clases. Todos estos nombres especiales comienzan y terminan con un doble guion bajo. Si en cualquier clase definida en Python se define un método (una función) llamada `__init__`, éste se interpreta como el método constructor de la clase. Los métodos `__str__` y `__repr__` se utilizan para imprimir los valores de los atributos de los objetos (es decir, para

convertirlos en cadenas de caracteres equivalentes…el primero funciona para imprimir los valores de objetos aislados, mientras que el segundo funciona para imprimir los valores de objetos del tipo correspondiente cuando están agrupados dentro de una lista u otra colección). El método `__lt__`, que es una abreviatura de "less than", se utiliza para comparar objetos cuando éstos se usan como operandos del operador relacional `<` (y pueden definirse también `__gt__` y algunos otros métodos parecidos), regresando un valor entero que en teoría indica qué tan apartados están entre sí los objetos que se están comparando (como se suele hacer en Java cuando se incluye un método `compareTo` dentro de la definición de alguna clase). El método `__eq__` se usa para determinar la igualdad (en cuanto a los valores de los atributos importantes) de dos objetos y regresa un valor booleano como respuesta. Hay varios otros "métodos mágicos" como estos (que se ejecutan automáticamente en ciertas circunstancias, en caso de haber sido definidos o heredados de alguna clase, como por ejemplo la clase predefinida `Object`, que es equivalente a la clase `Object` de Java y funciona de igual manera, es decir cualquier clase que se defina hereda de la clase `Object` automáticamente, sin tener que indicarlo explícitamente) en Python, pero éstos son los principales. En Python se usa la palabra reservada `self` como equivalente de `this` en Java. Reiterando lo que ya se vio, el valor `None` en Python (el único valor posible del tipo `None`) es equivalente al valor `null` en Java (es decir, representa la ausencia de un objeto en memoria). La instanciación de una clase en Python consiste en asignarle a un identificador el resultado de invocar al constructor de la clase, lo cual se logra colocando del lado derecho de la asignación el nombre de la clase y entre paréntesis los valores que se deseen enviar como argumentos al método constructor correspondiente. Un ejemplo de todo lo anterior para definir una clase que no tiene superclases (excepto `Object`, la herencia de la cual ocurre implícitamente, automáticamente, igual que en Java) en Python es (se incluye un ejemplo de una lista aquí, aunque las listas se explicarán en detalle más adelante):

```
#Definición de una clase Entero con un atributo "valor" (cuyo valor estoy
#suponiendo que será un número entero):
class Entero:
    #No se tiene que listar el atributo además de los métodos (aunque
    #se podría, por ejemplo si se desea asignarle un valor por omisión)...
    #en este ejemplo el atributo queda implícito por la forma en la que está
    #definido el constructor.
    #Constructor:
    def __init__(self,valor):
        self.valor=valor
    #Equivalente a toString:
    def __str__(self):
        return str(self.valor)
    #Equivalente a toString pero usado cuando se imprime una lista de objetos
    #(y en otros contextos)...si no se definiera este método, abajo no se
    #vería información útil/amigable al hacer "print(lista1)":
    def __repr__(self):
        return str(self.valor)
    #Sirve para poder hacer algo parecido al compareTo (método "less than"):
    def __lt__(self,other):
        return self.valor<other.valor

#Creación e impresión de varias instancias (y una lista de instancias)
#de la clase Entero:
obj1=Entero(1)
obj2=Entero(2)
```

```
obj3=Entero(3)
obj4=Entero(4)
obj5=Entero(5)
obj_14=Entero(-14)
print(obj2)
print(obj4)
lista1=[obj1,obj2,obj3,obj4,obj5,obj_14]
print(lista1)
```

Un ejemplo del comienzo de la definición de una clase `FacultadMenor` en Python que hereda de dos clases `Alumno` y `Empleado` (además de `Object`) es:

```
class FacultadMenor(Alumno,Empleado):
    #Cuerpo de la clase…
```

2. Python—fundamentos parte 2:

Estructuras algorítmicas condicionales y repetitivas:
En Python las estructuras algorítmicas condicionales/selectivas que están disponibles son el `if` simple y el `if-else`. En Python no hay un condicional múltiple/multivía como `switch`, aunque sí hay `if-elif-elif-…-else` (este último es equivalente a tener una "cascada" de `if-else-if-else-if-else` anidados, pero sin tener que introducir tantos niveles de sangrías en el código fuente). En Python las estructuras algorítmicas cíclicas/repetitivas que están disponibles son `while` y `for` (pero, al contrario de Java, donde el `for` es un caso especial del `while`, el `for` en Python es más como el "foreach"—es decir, un `for` para estructuras iterables—de Java y otros lenguajes). En Python no hay `do-while`. En Python es necesario usar paréntesis cuando se definen y cuando se invocan funciones/métodos, al igual que en Java, pero al contrario de Java no es necesario usar paréntesis al especificar las condiciones de las operaciones selectivas o repetitivas. Algunos ejemplos (entre los cuales se encuentran algunas listas y algunas cadenas de caracteres, las cuales son tipos de colección que se explicarán en detalle más adelante) son:

```
a=5
b=10
if a<b:
    print("a es menor que b.")

if a<b:
    print("a es menor que b.")
else:
    print("b es menor que a.")

#Ejemplos de "in" usado para consultar membresía:
letra="j"
if letra in ["a","e","i","o","u"]:
    print("Vocal")
elif letra in ["b","m","p"]:
    print("Consonante bilabial")
elif letra in ["f","v"]:
    print("Consonante labiodental")
elif letra in ["g","k","w","y"]:
    print("Consonante dorsal")
else:
    print("Otra consonante")

#El último ejemplo equivale a:
if letra in ["a","e","i","o","u"]:
    print("Vocal")
else:
    if letra in ["b","m","p"]:
        print("Consonante bilabial")
    else:
        if letra in ["f","v"]:
            print("Consonante labiodental")
        else:
            if letra in ["g","k","w","y"]:
                print("Consonante dorsal")
            else:
                print("Otra consonante")
```

```
#Imprime 4 3 2 1 0:
h=4
while h>=0:
    print(h)
    h=h-1

#Ejemplos de "in" usado para extraer los elementos de colecciones:
#Imprime uno por uno los elementos de una lista:
for i in ["alfa","beta","gama"]:
    print(i)

#Imprime sólo las vocales de una palabra (cadena de caracteres):
for letra in "elefante":
    if letra in ["a","e","i","o","u"]:
        print(letra)

#Imprime los valores enteros de 0 a 5 (inclusive):
for val in range(6):
    print(val)

#Imprime los valores enteros de 2 a 5 (inclusive):
for val in range(2,6):
    print(val)

#Imprime los valores enteros de 2 a 15 (inclusive) de tres en tres:
for val in range(2,16,3):
    print(val)
```

Más detalles sobre la definición de funciones:
En Python no existe la sobrecarga de funciones (no puede haber varios métodos, varias funciones, con el mismo nombre). Sin embargo, en Python se puede lograr un efecto parcialmente parecido al que se logra a través de la sobrecarga en Java gracias a la existencia de parámetros optativos (y valores por omisión para dichos parámetros) en las funciones. Cualquier parámetro para el cual se define un valor por omisión automáticamente se vuelve un parámetro optativo, pero hay que tener cuidado al definir una función con parámetros optativos: si hay *n* parámetros optativos, éstos tienen que ser los últimos *n* en la lista de parámetros en la definición de la función. Un ejemplo:

```
def tarjeta_de_credito(nombre,numero,limite_de_credito=6000):
    #Cuerpo de la función incluyendo un "return"

a=tarjeta_de_credito("Chandrika Bandaranaike Kumaratunga",7828932988384988)
b=tarjeta_de_credito("Neymar",1283928366129473,8)
```

La tarjeta de crédito de Chandrika Bandaranaike Kumaratunga terminará teniendo un límite de crédito de 6000 unidades monetarias, mientras que la tarjeta de crédito de Neymar tendrá un límite de crédito de 8 unidades monetarias. En Java esto se hubiera logrado a través de la definición de dos métodos llamados `tarjeta_de_credito` (o más bien `tarjetaDeCredito`), uno de los cuales hubiera recibido dos argumentos y el otro tres.

Otro aspecto de las funciones en Python es la invocación de funciones mediante la identificación explícita, por parte del programador, de cada argumento. Esto permite no tener que recordar el orden en el que se deben especificar los argumentos (pero, a cambio, se tiene que saber cómo se llamaron los parámetros cuando se definió la función en cuestión). En inglés este uso se conoce como "keyword arguments" (o, de forma abreviada, "kwargs"). Se puede invocar a cualquier función haciendo uso de la identificación de los argumentos (donde "identificar los argumentos" implica especificar explícitamente a qué parámetro en la definición de la función corresponde cada valor enviado como argumento). Un ejemplo:

```
def funcion(mensaje,valor_num1,valor_num2):
    valor_final=valor_num1+valor_num2
    print("El mensaje es: "+mensaje+" y la suma es:",valor_final)
funcion("hola mundo",5,6) #Imprime correctamente
#Pero la siguiente invocación marca error por no colocar los argumentos en las
#posiciones correctas (lo cual llevaría a Python a tratar de sumar o
#concatenar un entero, 6, con una cadena de caracteres, en la 1ª línea de
#código dentro del cuerpo de la función arriba, que es donde ocurre el error
#en tiempo de ejecución):
#funcion(5,6,"hola mundo")
#
#Por otra parte en el siguiente ejemplo las dos invocaciones a la función
#funcionan perfectamente aunque no se respeta el orden en el que se
#definieron los parámetros a la hora de invocar la función por segunda vez,
#debido a que la segunda invocación usa palabras clave para identificar los
#argumentos reales (i.e., para indicar cómo deben ser interpretados):
def funcion2(mensaje,valor_num1,valor_num2):
    valor_final=valor_num1+valor_num2
    print("El mensaje es: "+mensaje+" y la suma es:",valor_final)
funcion2("hola mundo",5,6) #Imprime correctamente
funcion2(valor_num1=5,valor_num2=6,mensaje="hola mundo") #También imprime
                                                         # correctamente
```

Las funciones en Python pueden regresar múltiples valores "al mismo tiempo", pero para no perder información hay que tener cuidado de recibir esos múltiples valores dentro de múltiples variables en el momento de invocar a una función que haya sido definida así. Internamente la forma en la que realmente se maneja esto de los múltiples valores de retorno es a través de la creación implícita de una tupla (las tuplas son uno de los tipos de colección que tiene Python de manera predefinida, y se verán más adelante). Una función que "regresa múltiples valores" en realidad regresa una sola tupla que agrupa los "múltiples valores". Por lo tanto también se puede recibir el resultado de invocar a una función así en una tupla (y luego extraer sus múltiples componentes, donde/conforme sea necesario), en lugar de recibir el resultado dentro múltiples variables. Un ejemplo:

```
#Regresa dos valores: el promedio de la lista de valores numéricos que se le
#entrega y la cantidad de valores que tuvo dicha lista:
def funcion3(lista):
    suma=0.0
    n=len(lista)
    for i in range(0,n):
        suma=suma+lista[i]
    prom=suma/n
    return prom,n
    #Hubiera funcionado exactamente igual si se hubiera hecho:
    #return (prom,n)
#Se puede invocar la función así:
p,num=funcion3([1,2,3,4])
print("Promedio:",p,"Cantidad:",num)  #Imprime: Promedio: 2.5 Cantidad: 4
#También se puede invocar así (con una tupla explícita):
tup=funcion3([1,2,3,4])
print("Promedio:",tup[0],"Cantidad:",tup[1])  #Imprime lo mismo que arriba.
print(tup)  #Imprime (2.5, 4)
```

También es posible en Python definir funciones que reciben cantidades variables de argumentos, sin especificar cuántos deben ser. La forma en la que funciona dicho mecanismo es interpretando "el argumento" como algo iterable (y dentro del cuerpo de la función se puede hacer la iteración, es decir definir un ciclo que analiza/usa/interpreta por separado cada uno de los argumentos que hayan entrado a la función en una invocación específica a la misma). Se pueden encontrar detalles y ejemplos en:

https://www.geeksforgeeks.org/args-kwargs-python/

Entrada de datos a un programa en Python:
Ya hemos visto a través de algunos de los ejemplos anteriores que la escritura/salida de datos en Python a la consola se logra a través de instrucciones `print`. La instrucción inversa, que permite leer/ingresar datos de la consola (en realidad, del teclado) en Python es `input`. Aquí se ven algunos ejemplos:

```
print("Ingrese un número: ")
x=input()
print(x)  #Imprime 4 si se ingresa un 4

#Tiene el mismo efecto que:
x=input("Ingrese un número: ")
print(x)  #Imprime 4 si se ingresa un 4

#Es posible (según lo que se desee/necesite hacer con el dato leído en el
#resto del programa) que se tenga que hacer un "cast" al valor leído:
x=float(input("Ingrese un número: "))
print(x)  #Imprime 4.0 si se ingresa un 4
```

Manejo de excepciones en Python:
Python es un lenguaje que permite que el programador incluya el manejo de excepciones (errores que ocurren en tiempo de ejecución) dentro de sus programas, como Java. En lugar del `throw` de Java, en Python se usa la palabra reservada `raise`. En lugar de los bloques `try-catch` de Java,

en Python los bloques equivalentes se llaman `try-except`. Aparte de eso, los funcionamientos de los mecanismos relacionados con las excepciones son muy parecidos en los dos lenguajes. Un ejemplo breve es:

```
#Clase que representa una excepción no predefinida, cola vacía:
class EmptyQueueException(RuntimeError):
    def __init__(self, arg):
        self.args = arg

#Clase que usa una lista de Python para implementar el concepto de cola.
#Supone que se agregan los elementos a la derecha (es decir, en el extremo
#con los índices más grandes, usando el método "append") de la lista
#conforme va creciendo (y por lo tanto se deben eliminar del lado
#izquierdo, usando el método "pop").  Las listas de Python (y sus métodos)
#se explicarán en detalle más adelante:
class ListQueue:
    #Constructor (inicialmente va a estar vacía la cola):
    def __init__(self):
        self.datos=[]
    def enqueue(self,dato):
        self.datos.append(dato)
    def dequeue(self):
        if(not self.is_empty()):
            res=self.datos[0]
            self.datos.pop(0)
            return res
        else:
            raise EmptyQueueException("La cola está vacía...")
#            print("Cola vacía...no se puede eliminar un dato.")
    def first(self):
        if(not self.is_empty()):
            return self.datos[0]
        else:
            raise EmptyQueueException("La cola está vacía...")
#            print("Cola vacía...no existe ningún dato.")
    def is_empty(self):
        return self.datos==[]

#Pruebas:
cola1=ListQueue()  #Se crea vacía la cola.
#El try-except termina imprimiendo un mensaje de error (es decir, se
#ejecuta el "except" porque no funciona el intento ("try") de eliminar un
#dato de la cola:
try:
    cola1.dequeue()
except EmptyQueueException:
    print("Error...")
cola1.enqueue(33)  #Ahora sí se agrega un dato a la cola.
#Se imprime el primer dato de la cola, sin alterar su contenido:
print(cola1.first())
#Se elimina un dato de la cola (no lo puse dentro de un "try" porque sé que
#va a terminarse de ejecutar la operación exitosamente):
cola1.dequeue()
print(cola1.is_empty())  #Se imprime "True" porque quedó vacía la cola.
```

3. Colecciones en Python:

Hay seis tipos predefinidos de colecciones en Python. La siguiente tabla muestra un resumen de algunas de sus características (otra característica importante—si el tipo de colección permite el almacenamiento de valores de distintos tipos o no—no se incluye en la tabla pero se discute más abajo, en las secciones individuales sobre los distintos tipos de colección):

Tipo de colección (y su nombre en Python):	**¿Mutable (modificable)?**	**¿Sus elementos mantienen su orden?**	**Ejemplo de notación:**
Cadenas de caracteres (`str`):	No	Sí	`"Este texto no dice nada"`
Listas (`list`):	Sí	Sí	`["Susana", 14.17, True]`
Tuplas (`tuple`):	No (pero sus componentes mutables sí)	Sí	`("Juan", 18, "Alumno")`
Conjuntos (`set`):	Sí	No	`{"AeroMéxico", "Interjet", "Volaris", "VivaAerobús"}`
Conjuntos congelados (`frozenset`):	No	No	`{"AeroMéxico", "Interjet", "Volaris", "VivaAerobús"}`
Diccionarios (`dictionary`):	Sí	No	`{"Coches" : ["VW", "Ford", "Renault", "Toyota"], "Felinos" : ["León", "Puma", "Leopardo", "Tigre"]}`

Para cualquiera de estas colecciones, se puede consultar cuántos elementos tiene una instancia mediante la función predefinida `len()` y se puede crear una copia de una instancia mediante la función predefinida `copy()`. También es posible convertir de una colección a otra, con una posible pérdida de información o invento de información, dependiendo de la dirección de la conversión, a través de un "cast" (por ejemplo, si `x` es una lista instanciada y se desea convertirla a un conjunto, llamándolo `k`, se puede hacer `k=set(x)` y se pierde el orden entre sus valores, así como la posibilidad de que haya valores duplicados). Como se puede ver en la tabla, algunas de las colecciones de Python son inmutables (una vez creado el objeto, el contenido de éste ya no se puede modificar—lo más que se puede hacer es crear una copia y en el momento de hacerla incorporarle cambios a la copia) y otras son mutables. Esto tiene implicaciones en cuanto a la eficiencia de su uso, tanto en el tiempo que se van a tardar las operaciones que se le apliquen a las colecciones como en la cantidad de memoria que utilizan. En algunas de estas colecciones sus elementos tienen un orden específico (es decir, esos tipos de colecciones son secuencias), pero en otras no. En las secuencias se puede acceder a los elementos individuales a través de índices

enteros que representan posiciones (índices), o se puede hacer un acceso a alguna subparte de la colección mediante el uso de "rebanadas" (slices), las cuales se basan en el uso de índices límite y se discuten más adelante. En Python el índice del primer elemento de cualquier colección ordenada (secuencia) es el número 0 (pero también hay índices negativos…ver abajo). Los índices numéricos no se pueden usar en Python para acceder a los elementos de las colecciones que no son secuencias. Vamos a presentar cada una de las colecciones de Python en mayor detalle.

Cadenas de caracteres:
Las cadenas de caracteres ("textos") constantes en Python se pueden expresar mediante el uso de comillas dobles o comillas sencillas al indicar los caracteres que le pertenecen, indistintamente (tienen la misma interpretación), aunque no se pueden emparejar comillas sencillas con dobles en la especificación de la misma cadena de caracteres. Dado que Python permite comillas sencillas o dobles para las cadenas y no distingue entre *char* y *String*, en Python `'c'` es lo mismo que `"c"`, al contrario de en Java. Las dos siguientes asignaciones son válidas y son equivalentes (también hay otras variantes posibles, pero éstas son las básicas):

```
frase="hola mundo"
frase='hola mundo'
```

Mientras que el siguiente intento es un error de sintaxis:

```
frase="hola mundo'
```

La idea detrás de esta flexibilidad es poder fácilmente incluir comillas (ya sea sencillas o dobles) dentro de las cadenas de caracteres (sin causarle problemas de entendimiento al traductor), como se muestra en el siguiente ejemplo:

```
print('Él le dijo "cuídate mucho" a su mamá.')
```

Las cadenas de caracteres en Python son secuencias inmutables de caracteres. Una característica de Python es el uso de índices negativos: el último símbolo de una cadena de caracteres (es decir, el símbolo que está hasta la derecha de la cadena) tiene el índice -1, el penúltimo símbolo tiene el índice -2, y así sucesivamente. El primer símbolo de una cadena con *n* caracteres (el símbolo que está hasta la izquierda de la misma) tiene el índice negativo *–n*, aparte de tener el índice 0. Otra característica de Python, conocida como "slicing" en inglés, es el uso de "rebanadas", que son subcadenas especificadas a través del índice de inicio (donde "inicio" implica el extremo izquierdo del texto que formará parte de la rebanada) y el índice de terminación (donde "terminación" implica el extremo derecho de la rebanada) de la subcadena. El índice de inicio debe ser incluyente (debe ser el índice del primer símbolo que pertenece a la cadena que se desea incluir en la rebanada) pero el índice de terminación debe ser excluyente (debe ser el índice del elemento subsecuente al último elemento que se desea considerar para su inclusión en la subcadena). Estas subcadenas no son objetos distintos en memoria, sino que se accede a sólo una subparte del objeto completo basándose en la especificación de la rebanada (la cual se hace usando un "dos puntos" `:`). La especificación de la rebanada puede omitir uno de los dos índices (ya sea el de izquierdo o el derecho) si se desea comenzar en el primer símbolo, o terminar en el último, respectivamente, de la cadena. Finalmente, también se puede definir una tercera

componente de la rebanada, el "salto" ("stride" en inglés). El salto es una indicación de la diferencia en posiciones entre los caracteres que se desea incluir en la subcadena), y para especificarlo también se separa de las primeras dos componentes a través de un `:`. Si el salto es negativo, esto indica que se desea tener el resultado en orden inverso (es decir, se procesan los caracteres de la cadena original de derecha a izquierda, por lo que el primer índice especificado en la rebanada debe ser el extremo derecho (sigue siendo el inicio) de la misma, aunque sigue siendo incluyente, y el segundo índice especificado debe ser el extremo izquierdo de la rebanada (sigue siendo el final), aunque sigue siendo excluyente. Cuando se especifica un salto, las primeras dos componentes de la especificación de la rebanada se pueden omitir (si se desea considerar todos los caracteres de la cadena para su inclusión en la rebanada), aunque se siguen necesitando los separadores `:` correspondientes. Los tres posibles valores en la especificación de la rebanada deben ser números enteros.

Una representación gráfica de los índices en Python se muestra a continuación para la cadena `"hola mundo"`. Los índices negativos se muestran arriba, el contenido de las distintas posiciones dentro de la cadena en medio y los índices positivos (tradicionales) abajo:

-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
h	**o**	**l**	**a**		**m**	**u**	**n**	**d**	**o**
0	1	2	3	4	5	6	7	8	9

Vamos a suponer que ya se hizo:

```
frase="hola mundo"
```

Esta frase de prueba tiene 10 caracteres numerados de izquierda a derecha del 0 al 9 (o del -10 al -1). Entonces:

```
print("1",frase[0])  #Imprime "h", el elemento 0
print("2",len(frase))  #Imprime 10
#Provocaría el lanzamiento de una excepción de tipo IndexError (con un mensaje
#de error: "string index out of range"):
#print(frase[10])
#Provocaría el lanzamiento de una excepción de tipo TypeError (con un mensaje
#de error: "'str' object does not support item assignment") debido a que las
#cadenas son inmutables:
#frase[2]="z"
print("3",frase[-len(frase)])  #Imprime "h", el elemento -10, que es
#equivalente al 0.
print("4",frase[0:6])  #Imprime "hola m", los elementos 0 al 5 de la frase
print("5",frase[-10:-4])  #Imprime "hola m", los elementos -10 al -5 (que
#equivale al 5)
#Es muy importante el orden en el que se especifican los límites de la
#"rebanada":
print("6",frase[-4:-10])  #Imprime "" (cadena vacía) puesto que el
#índice izquierdo no puede estar más a la derecha que el índice derecho
#de la "rebanada" para que exista una subcadena
print("7",frase[:6])  #Imprime "hola m", los primeros elementos hasta el
#número 5 de la frase
print("8",frase[6:])  #Imprime "undo", los últimos elementos desde el
#número 6 de la frase
```

```
#Ahora algunos ejemplos con "salto" (tercera componente en la
#especificación de la "rebanada"):
print("9",frase[0:6:2])  #Imprime los primeros "seis" caracteres
#saltándose uno de cada dos (es decir, realmente imprime sólo
#tres caracteres, "hl ")
print("10",frase[-10:-4:3])  #Imprime los caracteres del 0
#(equivalente a -10) al 5 (equivalente al -5), saltándose dos de
#cada tres, por lo tanto imprimiendo "ha"
print("11",frase[::4])  #Imprime uno de cada cuatro caracteres de
#principio a fin de la frase (por lo tanto imprimiendo "h d")
#Si el salto es negativo, entonces se procesa de derecha a izquierda la
#cadena de caracteres, y entonces cambian las interpretaciones de las
#primeras dos partes de la especificación de la rebanada: ahora la primera
#es el extremo derecho de la rebanada (sigue siendo el inicio, y sigue siendo
#incluyente) y la segunda es el extremo izquierdo de la rebanada (sigue
#siendo el final, y sigue siendo excluyente), como muestran los siguientes
#ejemplos:
print("12",frase[::-2])  #Imprime uno de cada dos caracteres del
#fin al principio de la frase debido al salto negativo ("onmao").
print("13",len(frase[::-2]))  #Cuenta cuántos caracteres hay en la
#rebanada anterior (5).
print("14",frase[-4:-10:-3])  #Imprime, debido al salto negativo, los
#caracteres del -4 (equivalente al 6) al -9 (equivalente al 1),
#saltándose dos de cada tres, por lo tanto imprimiendo "ua".
print("15",frase[6:1:-3])  #Imprime exactamente lo mismo ("ua") que el
#ejemplo anterior debido a las equivalencias de los índices ("ua").
```

Algunos métodos predefinidos para el manejo de cadenas de caracteres en Python son:

- `count()`: cuenta cuántas veces aparece una cadena (que puede consistir de uno o varios símbolos) dentro de la cadena a la que se le haya aplicado el método.
- `find()`: encuentra el índice de la primera aparición (contando de izquierda a derecha) de una cadena dentro de la cadena a la que se le haya aplicado el método (también se puede especificar que se desea la segunda, tercera, etc., aparición). Un método parecido es `index()`, con la diferencia de que `index()` lanza una excepción en caso de no encontrar lo que se buscaba en ninguna parte de la cadena, mientras que `find()` regresa un -1 para indicar la misma situación.
- `isdigit()`, `isalpha()`, `isalnum()`, `isupper()`, `islower()`, `isspace()`: predicados que indican si la cadena de caracteres a la que se le haya aplicado alguno de estos métodos consiste exclusivamente de dígitos, letras, dígitos y letras, mayúsculas, minúsculas, o espacios, respectivamente, o no.
- `istitle()`: predicado que indica si en la cadena de caracteres a la que se le haya aplicado el método cada palabra empieza con una letra mayúscula o no.
- `title()`, `capitalize()`, `upper()`, `lower()`, `swapcase()`: métodos que regresan nuevas cadenas de caracteres (también inmutables, igual que las cadenas a las que se les hayan aplicado los métodos) en las que, respectivamente, se haya convertido la primera letra de cada palabra en mayúscula, se haya convertido la primera letra de toda la cadena en mayúscula, se haya convertido todas las letras de la cadena en mayúsculas, se haya convertido todas las letras de la cadena en minúsculas y se hayan convertido las minúsculas en mayúsculas y viceversa.
- `partition()`, `split()`: métodos que dividen la cadena de caracteres a las que se le haya aplicado alguno de los métodos de acuerdo con el separador especificado como

argumento, en el primer de los casos guardando el resultado en una tupla (y realizando una sola partición), y en el segundo de los casos guardando el resultado en una lista (y realizando todas las particiones necesarias dependiendo de la cantidad de apariciones dentro de la cadena del separador especificado como argumento). (Se discuten las tuplas y las listas en mayor detalle abajo.)

- `strip()`, `replace()`: métodos que regresan nuevas cadenas de caracteres en las que, respectivamente, se hayan eliminado algunos caracteres de la cadena a la que se haya aplicado el método o reemplazado algunos caracteres por otros dentro de la cadena a la que se haya aplicado el método.
- `startswith()`, `endswith()`: predicados que indican si la cadena de caracteres a la que se le haya aplicado el método empieza o termina con la subcadena que se haya especificado como argumento, respectivamente.

Algunos ejemplos de uso (y sutilezas del funcionamiento) de algunos de los métodos anteriores junto con los comentarios correspondientes son los siguientes (seguimos suponiendo que ya se hizo `frase="hola mundo"`):

```
print(frase.count("mu"))  # Imprime 1 porque sólo aparece la subcadena "mu"
#una vez dentro de la frase.
print(frase.count("o"))  # Immprime 2 porque la subcadena "o" aparece dos
#veces dentro de la frase.
print(frase.count("z"))  # Imprime 0 porque la subcadena "z" no aparece ni una
#sola vez dentro de la frase.
print(frase.count(""))  # Imprime 11 porque la cadena vacía cabe una vez en
#cada uno de los 11 _huecos_ que hay entre las diez letras que forman la
#frase.

print(frase.find("mu"))  # Imprime 5 porque a partir del índice 5 de la frase
#es que aparece la subcadena "mu".
print(frase.find("o"))  # Imprime 1 porque la primera "o" de la frase está en
#el índice 1.
print(frase.find("o",2))  # Imprime 9 porque la segunda "o" de la frase
#está en el índice 9.
print(frase.find("o",3))  # Imprime 9 porque, aunque no hay una tercera "o"
#dentro de la frase, la última (que resulta ser la segunda) "o" está en el
#índice 9.
print(frase.find("z"))  # Imprime -1 para indicar que no aparece la subcadena
#"z" en ninguna parte de la frase.

#Muy parecido al bloque de instrucciones anterior, pero al usar el método
#"index" en lugar de "find" se lanza una excepción en lugar de regresar un -1
#cuando no aparece la subcadena especificada en el primer argumento dentro de
#la frase (debido a lo cual se comentó la última instrucción):
print(frase.index("mu"))
print(frase.index("o"))
print(frase.index("o",2))
print(frase.index("o",3))
#print(frase.index("z"))

print(frase.islower())  # Imprime True porque todas las letras de la frase son
#minúsculas.
print("bla bla 123 bla, bla".islower())  #Imprime True porque todas las letras
#de esta nueva frase son minúsculas (y no le hace caso a los caracteres que no
```

```
#son alfabéticos).

print(frase.istitle())  # Imprime False porque no todas las palabras que
#forman parte de la frase (de hecho, en este caso ninguna de ellas) empieza
#con mayúscula.
print("Stairway To Heaven".istitle())  # Imprime True porque todas las
#palabras que forman parte de la frase empiezan con mayúscuas.
print("Stairway 123 To3 Heaven".istitle())  # Imprime True porque todas las
#palabras que forman parte de la frase empiezan con mayúsculas (si empiezan
#con letras...y no se les hace caso a las que empiezan con otros catacteres).
print("Stairway 123 to3 Heaven".istitle())  # Imprime False porque una de las
#palabras que forman parte de la frase (en este caso la tercera) no comienza
#con mayúscula.

print(frase.title())  # Imprime "Hola Mundo" porque hace que todas las
#palabras que formen parte de la frase empiecen con mayúsculas.
print(frase.capitalize())  # Imprime "Hola mundo" porque hace que la primera
#palabra que forma parte de la frase empiece con mayúscula (si es que empieza
#con una letra).
print("123 hola mundo".capitalize())  # Imprime "123 hola mundo" porque hace
#que la primera palabra que forma parte de la frase empiece con mayúscula
#(pero en este caso la primera palabra no empieza con letra, así es que la
#deja, al igual que el resto de la frase, tal cual).
```

Todos los operadores relacionales (<, >, <=, >=, ==, !=) se pueden utilizar para comparar cadenas de caracteres en Python. Dos operadores de concatenación que existen en Python son + (para una sola concatenación) y * (para múltiples concatenaciones, donde el segundo argumento debe ser un número entero positivo). Entonces:

```
cad="hola mundo"
print(cad<"hola mundo ")  #Imprime: True
print(cad=="hola mundo")  #Imprime: True
print(cad>="gola mundo")  #Imprime: True
print("a"!=cad)  #Imprime: True
print("a"==cad)  #Imprime: False
print(cad+cad)  #Imprime: "hola mundohola mundo"
print(cad+" cruel")  #Imprime: "hola mundo cruel"
print(cad*5)  #Imprime: " hola mundohola mundohola mundohola mundohola mundo"
```

Listas:

Las listas en Python son secuencias (el orden es importante) mutables de elementos que pueden ser heterogéneos. Para escribir el contenido de una lista se separan los elementos por comas y se incluyen dentro de un par de corchetes. Se usan índices entre corchetes para acceder a los elementos individuales de una lista. Los índices pueden ser positivos o negativos, siguiendo las mismas reglas que con las cadenas de caracteres. Con las listas se pueden usar los operadores + y *, con sintaxis y semántica parecida a las de los mismos operadores aplicados a cadenas de caracteres (ver arriba) o tuplas (ver abajo), así como la idea de las rebanadas. Debido a que las listas son mutables, al contrario de las cadenas de caracteres, las rebanadas de listas sí crean sublistas distintas (objetos distintos en memoria, con una copia de algunos de los elementos de la lista original), al contrario de lo que ocurre con las cadenas de caracteres, donde las rebanadas simplemente son "vistas" hacia algunos de los elementos de la cadena original, sin hacer copias de ellos dentro de la memoria. Esto tiene implicaciones en cuanto a la eficiencia (tanto en el uso

de la memoria como del tiempo del procesador) y en cuanto al impacto de las modificaciones que se realicen en las listas (o sublistas creadas a través del mecanismo de las rebanadas). Algunos ejemplos de rebanadas con listas son:

```
lista=[100,200,300,400,500,600]
print(lista[3])  #Imprime 400 (el valor en el índice 3 de la lista).
print(lista[:3])  #Imprime los valores entre el inicio (índice 0) y el
#índice 2 de la lista (es decir, imprime [100,200,300]).
print(lista[3:])  #Imprime los valores entre el índice 3 y el final (índice 5)
#de la lista (es decir, imprime [400,500,600]).
print(lista[::2])  #Imprime de dos en dos los valores desde el inicio
#(índice 0) hasta el final de la lista (es decir, imprime [100,300,500]).
print(lista[::-2])  #Imprime de dos en dos los valores desde el último de
#la lista hasta el primero (es decir, imprime [600,400,200]).
print(lista[1:4])  #Imprime la rebanada que consiste de los valores que
#están entre el índice 1 y el índice 3 de la lista (es decir, imprime
#[200,300,400]).
print(lista[-5:-2])  #Imprime la rebanada que consiste de los valores que
#están entre el índice -5 (que equivale a 1) y el índice -3 (que equivale a
#4).  Es decir, imprime la misma rebanada que se imprime en el ejemplo
#anterior: [200,300,400].
print(lista[1:4][1])  #Imprime el segundo de los elementos (el que está en el
#índice 1) de la rebanada especificada en la instrucción anterior (es
#decir, imprime 300).
lista[1:4][1]=800  #Cambiamos el valor del segundo elemento de la rebanada
#especificada en las dos instrucciones anteriores (de 300 a 800).
print(lista)  #Imprime la lista completa (es decir, imprime [100,200,300,
#400,500,600]), con lo cual se puede ver que la modificación del segundo
#elemento de la rebanada no afectó la lista original, por lo que la rebanada
#se almacenó por separado en otra parte de la memoria de la computadora (no
#fue una "vista" hacia los elementos originales de la lista, sino que se creó
#una sublista con los elementos especificados por la rebanada, y fue en esta
#sublista donde ocurrió el cambio al valor del segundo elemento).
print(lista[1:4])  #Se imprime otra vez la rebanada que consiste de los
#valores que están entre el índice 1 y el índice 3 de la lista (es decir,
#imprime [200,300,400]), con lo cual podemos ver que cada vez que se pide
#una rebanada, ésta se construye otra vez (en otra parte de la memoria,
#como se mencionó en el punto anterior), puesto que en caso contrario
#hubiéramos tenido un 800 en lugar de un 300 como segundo elemento de la
#rebanada después del cambio que le hicimos anteriormente.
#En la siguiente secuencia de instrucciones podemos ver el mismo efecto, pero
#colocando explícitamente la rebanada en una variable distinta (por lo tanto
#en una parte distinta de memoria) antes de realizar el cambio, y viendo
#que dicho cambio ocurre sólo en la sublista especificada por la rebanada, a
#la cual afecta permanentemente, no en la lista original, la cual nunca se
#ve afectada:
sublista=lista[1:4]
print(sublista)  #Imprime [200,300,400].
sublista[1]=800
print(sublista)  #Imprime [200,800,400].
print(lista)  #Imprime [100,200,300,400,500,600].
#Todas estas pruebas con los cambios a las sublistas creadas basándose en
#las especificaciones de las rebanadas no se podrían haber hecho con cadenas
#de caracteres, puesto que ellas son inmutables, al contrario de las listas...
#pero debido a que las cadenas de caracteres son inmutables, me imagino que
#las rebanadas en las cadenas no son copias del original, sino "vistas"
```

```
#construidas de una forma computacionalmente eficiente hacia algunos de los
#elementos de las cadenas originales, al contrario de lo que ocurre con las
#listas.
```

Algunos métodos útiles disponibles para procesar listas son:

- `index()`: regresa el índice de la primera aparición, analizando el contenido de la lista a la que se le aplique el método desde el índice 0 hacia índices más altos, uno por uno, del elemento indicado como argumento.
- `count()`: cuenta cuántas apariciones ocurren dentro de la lista a la que se le aplique el método del argumento especificado.
- `append()`: sirve para agregar un elemento, especificado como argumento, al final (al extremo derecho, al extremo de la lista con el índice más alto) de la lista a la que se le haya aplicado el método.
- `extend()`: sirve para agregar todos los elementos de un objeto iterable (por ejemplo, otra lista) especificado como argumento al final de la lista a la que se le haya aplicado el método.
- `insert()`: permite agregar un elemento, especificado como segundo argumento del método, a la lista a la que se le haya aplicado el método, tal que la nueva posición del elemento dentro de la lista sea en el índice que se haya especificado como primer argumento del método.
- `remove()`: elimina la primera aparición del elemento que se haya especificado como argumento dentro de la lista a la que se le haya aplicado el método (analizando el contenido de la lista desde el índice 0 hacia índices más altos).
- `pop()`: si se usa este método sin argumentos, se elimina y regresa el elemento final (es decir, el elemento en el índice más alto que existe) de la lista a la que se le haya aplicado el método. Una variante del método que requiere/recibe un argumento entero permite eliminar y regresar el elemento que esté en el índice especificado como argumento.
- `sort()`: utiliza un algoritmo de ordenamiento conocido como Timsort (que estratégicamente es un híbrido entre "mergesort" e "insertion sort") para ordenar los elementos de la lista a la que se le haya aplicado el método.
- `reverse()`: invierte el orden de los elementos de la lista a la que se le haya aplicado el método.

Existe un mecanismo en Python que en inglés se llama "list comprehension" para la generación automatizada de listas que sintácticamente es compacto y computacionalmente es eficiente (ya que permite especificar las condiciones que deben cumplir los elementos de una secuencia de valores sistemática para que se incluyan en la lista que se está generando, y sólo esos elementos que cumplen con las condiciones se incorporan). Un ejemplo es el siguiente:

```
x=[i for i in range(10) if i%2==0]  #El argumento de "range" es excluyente.
print(x)  #Imprime: [0, 2, 4, 6, 8]
```

El código anterior es más compacto que el siguiente (que da resultados equivalentes):

```
x=[]
for i in range(10):
    if i%2==0:
        x=x+[i]
print(x)  #Imprime: [0, 2, 4, 6, 8]
```

Como una extensión del concepto de las rebanadas, suponiendo que ya exista una lista también se pueden crear sublistas, o incluso sublistas modificadas, a través de filtros parecidos al del ejemplo de "list comprehension" arriba, como se muestra en el siguiente ejemplo:

```
lista1=[1,2,3,4]
lista2=[elem*2 for elem in lista1 if elem>1]
print(lista2)  #Imprime: [4,6,8]
```

Tuplas:
Las tuplas son agrupaciones inmutables de datos que pueden ser heterogéneos (separados por comas y escritos entre paréntesis) en Python. El orden en el que aparecen los datos es importante. Los paréntesis se pueden omitir al construir la tupla. Si se desea crear una tupla que contenga un solo valor, de todos modos se tiene que usar una coma. Ejemplos:

```
tupla1=(1,2)
tupla2=3,4
print(tupla1)  #Imprime: (1,2)
print(tupla2)  #Imprime: (3,4)
tupla3=5,
print(tupla3)  #Imprime: (5,)
```

Se pueden utilizar las rebanadas y los operadores `+` y `*`, con sintaxis y efectos similares a los que tienen en las cadenas de caracteres y listas, con las tuplas de Python. Debido a que las tuplas son inmutables, las rebanadas en las tuplas se comportan como las de las cadenas de caracteres: son vistas parciales hacia algunos de los elementos de la tupla original, sin hacer una copia de dichos elementos en memoria. También se puede acceder a los elementos individuales de las tuplas a través de índices especificados dentro de corchetes, igual que con las cadenas de caracteres y las listas. En realidad las tuplas no son 100% inmutables: cuando uno de los elementos de una tupla es mutable (por ejemplo, si es una lista), este elemento sí se puede cambiar (pero el resto de la tupla no). Por lo tanto:

```
tupla=([1],2,3)
tupla[0][0]=8  #El primer 0 indica que queremos acceder al primer
#elemento de la tupla, que resulta ser una lista…el segundo 0 indica que es el
#primer elemento de ese primer elemento (de la lista) el que queremos
#modificar.
print(tupla)  #Imprime: ([8],2,3)
```

Si se hubiera intentado realizar alguna operación parecida, pero para modificar cualquiera de los otros dos elementos de la tupla, hubiera ocurrido un error.

Conjuntos:
Los conjuntos en Python son colecciones mutables de datos que pueden ser heterogéneos, aunque los datos deben ser inmutables. Adicionalmente, no se puedan repetir elementos dentro de la colección. Los elementos de un conjunto se escriben separados por comas y agrupados entre llaves. Si se pide imprimir el contenido de un conjunto, los elementos de éste van a ser presentados por la computadora en un orden específico, pero a pesar de esto realmente no hay un orden entre los elementos del conjunto, lo cual implica que no se pueden usar índices (ni los demás fenómenos asociados con éstos, como rebanadas) para acceder a los elementos de un conjunto. Incluso, el orden en el que aparezcan los elementos de un conjunto podría variar si se pide varias veces imprimir el contenido del mismo conjunto. Los conjuntos son mutables en Python, pero cuando existe la necesidad de contar con conjuntos inmutables, o cuando existe la necesidad de manejar conjuntos cuyos elementos son mutables (por ejemplo, si son otros conjuntos, o si son listas) el lenguaje Python nos proporciona la existencia de "frozensets" (conjuntos congelados), los cuales no se discutirán en mayor detalle aquí (pero proporcionan operadores y métodos parecidos a los de los conjuntos "normales", discutidos a continuación).

En Python se pueden utilizar los operadores binarios `|`, `&`, `<`, `>`, `-` y `^` para manipular conjuntos (los dos operandos deben ser conjuntos), o se pueden utilizar los nombres de métodos equivalentes: `union()`, `intersection()`, `issubset()`, `issuperset()`, `difference()` y `symmetric_difference()`, respectivamente.

Algunos otros métodos útiles que se pueden usar para manipular conjuntos en Python son: `add()`, `remove()`, `pop()`, `isdisjoint()` y `clear()`. Algunos ejemplos de operaciones con conjuntos en Python son los siguientes:

```
a={1,2,3,4}
b={3,4,5,6}
c=a|b  #Unión
print(c)  #Imprime: {1,2,3,4,5,6}
print(a&b)  #Intersección (imprime: {3,4})
#La siguiente instrucción provocaría un error de sintaxis ("set object is
#not subscriptable") debido a que los conjuntos no tienen índices:
#print(a[1])
d={"a",3,4.5}
#El orden en el que se imprimen los elementos de un conjunto no necesariamente
#corresponde al orden en el que se agregaron dichos elementos:
print(d)  #Imprime: {3,"a",4.5}...a veces (pero puede imprimirse cualquier
#otra permutación de los mismos elementos).
#Prueba de que son mutables los conjuntos:
d.add("k")
print(d)  #Imprime: {"k",3,"a",4.5} o alguna permutación de los mismos
#elementos.
#Pruebas con el operador subconjunto (abierto y cerrado):
print(d<c)  #Imprime: False
print(d<=c)  #Imprime: False
```

```
conj1={1,2,3}
conj2={3,2,1}
print(conj1==conj2)  #Imprime: True (porque todos los elementos de "conj1"
#están en "conj2" y viceversa).
print(conj1<conj2)   #Imprime: False (porque "conj1" no es un subconjunto
propio de "conj2").
print(conj1<=conj2)  #Imprime: True (porque "conj1" sí es un subconjunto,
#aunque no sea un subconjunto propio, de "conj2").
```

Diccionarios:
Los diccionarios en Python son colecciones mutables de pares de elementos (estos pares se podrían visualizar como tuplas de dos elementos, pero no se usa la notación de tuplas para especificarlos) sin ningún orden en particular entre los pares (por lo que no hay índices numéricos ni sus fenómenos asociados como las rebanadas), de tal forma que en cada par el primer elemento es una llave/clave y el segundo es un dato. La llave/clave funciona como si fuera un índice (pero puede ser de cualquier tipo de dato inmutable, no necesariamente tiene que ser un número entero como un índice tradicional) y facilita el acceso al dato. Para definir el contenido de un diccionario se agrupan los datos entre llaves (como si el diccionario fuera un conjunto) y dentro de dichas llaves se usa la notación especial para diccionarios (basada en `:` como separador entre la llave/clave y el dato de cada par, y basada en `,` como separador entre pares). Esto se muestra en el siguiente código (en el cual las llaves/claves son números enteros en el primer diccionario-ejemplo y cadenas de caracteres en el segundo), el cual también ejemplifica algunos métodos útiles disponibles en los diccionarios:

```
d1={1:"cadena de caracteres",2:[12,22]}
d2={"animales":("elefante","leon","antílope"),"coches":("renault","volvo","for
d")}
print(d1.keys())  #Imprime: dict_keys([1,2])
print(d2.keys())  #Imprime: dict_keys(["animales","coches"])
print(d1.values())  #Imprime: dict_values(["cadena de caracteres",[12,22]])
print(d2.values())  #Imprime: dict_values([("elefante","leon","antílope"),
#("renault","volvo","ford")])
```

Aquí se ejemplifica el uso de los valores llave/clave como "índices" no numéricos en los diccionarios para acceder a los datos asociados (suponiendo que se han hecho las definiciones del código anterior):

```
print(d1[2])  #Imprime: [12,22]
print(d2["coches"])  #Imprime: ("renault","volvo","ford")
```

Algunos métodos útiles adicionales que se tienen disponibles para el manejo de diccionarios en Python son: `clear()`, `del()`, y `update()`.

Iteradores:
Se pueden realizar iteraciones (utilizando ciclos `for`, como hemos visto antes) sobre cualquier instancia de cualquier colección (sea una cadena de caracteres, una lista, una tupla, un conjunto, un conjunto congelado, o un diccionario) en Python. En el caso de las colecciones que no tienen un orden específico entre sus elementos (conjuntos, conjuntos congelados, o diccionarios), no hay forma de predecir el orden en el que se visitarán los elementos durante la iteración (y no necesariamente coincidirá con el orden en el que se agregaron los elementos en el momento de

crear la colección, ni tampoco necesariamente serían las visitas en el mismo orden si se itera desde cero varias veces sobre la misma instancia de colecciòn). Si no se desea iterar sobre una colección completa, sino únicamente mientras se cumpla (o no se cumpla) cierta condición, se puede iterar paso por paso usando un ciclo `while` con la condición apropiada según las necesidades. En este caso se debe usar la función predefinida `iter()` para obtener un objeto iterador que permita iterar sobre la colección de interés (la cual debe ser el argumento de la función). Después de esto, se puede usar la función predefinida `next()` para obtener el siguiente elemento de la iteración (dando como argumento el nombre del objeto iterador correspondiente), donde "el siguiente elemento" será el primero la primera vez que se utilice `next()`. La función `next()` puede lanzar una excepción (de tipo `StopIteration`) en caso de no haber ningún siguiente dato en la colección (ya sea porque la colección está vacía o porque ya se iteró una vez sobre cada uno de los datos—es decir, porque ya se terminó de visitar cada uno de los elementos de la colección durante la iteración). En este caso, para que no "truene" el programa hay que agregar un manejador de excepciones adecuado. El manejo de excepciones en Python (visto arriba) es muy parecido al de Java. Todo esto es necesario debido a que en Python no hay un equivalente a la función (el método) `hasNext()` de los iteradores, como el que hay en Java, que se pueda usar para evitar tener que incluir un manejador de excepciones apropiado (lo cual se lograría invocando a `next()` únicamente si se invoca a `hasNext()` y éste regresa un valor verdadero). Algunos ejemplos:

```
cadena="abcde"
lista=["a","b","c","d","e"]
tupla=("a","b","c","d","e")
conjunto={"a","b","c","d","e"}
conjunto_congelado=frozenset({"a","b","c","d","e"})
diccionario={"Coches":["Volvo","Ford","Toyota"],
             "Animales":["León","Elefante","Oso"],
             "Colores":["Rojo","Naranja","Amarillo","Verde","Azul","Violeta"]}

#Se puede iterar sobre una cadena:
print("Cadena:")
res=""
for x in cadena:
    res=res+x+" "
print(res)
print(len(res))

#Se puede iterar sobre una lista:
print("Lista:")
res=""
for x in lista:
    res=res+str(x)+" "
print(res)

#Se puede iterar sobre una tupla:
print("Tupla:")
res=""
for x in tupla:
    res=res+str(x)+" "
print(res)
```

```
#Se puede iterar sobre un conjunto, pero el orden en el que se visitarán
#sus elementos es arbitrario y no necesariamente coincide con el orden en
#el que se hayan agregado/indicado al crear el conjunto:
print("Conjunto:")
res=""
for x in conjunto:
    res=res+str(x)+" "
print(res)

#Se puede asignarle una posición (un índice) explícitamente a cada elemento
#de un conjunto mediante el uso de la función "enumerate", que crea tuplas
#índice-valor, pero de todas formas quedan en un orden arbitrario al iterar,
#así es que no es muy útil:
print("Enumerate de un conjunto, versión 1:")
res=""
en=enumerate(conjunto)
print(en)  #Objeto de tipo "enumerate"
for x in en:
    res=res+str(x)+" "
print(res)
#Aquí se extrae únicamente el valor de cada tupla índice-valor que resulte de
#la llamada a "enumerate" (si no se incluye la segunda llamada a "enumerate"
#no funciona por alguna razón...parece que se "corrompe" el resultado de la
#llamada anterior después de iterar sobre ella):
print("Enumerate de un conjunto, versión 2:")
res=""
en=enumerate(conjunto)
for z,y in en:
    res=res+str(y)+" "
print(res)

#Se puede iterar sobre un conjunto congelado, pero el orden en el que se
#visitarán sus elementos es arbitrario y no necesariamente coincide con el
#orden en el que se hayan agregado/indicado al crear el conjunto congelado,
#al igual que con los conjuntos "normales" (no congelados), así es que no es
#muy útil:
print("Conjunto congelado:")
res=""
for x in conjunto_congelado:
    res=res+str(x)+" "
print(res)

#Se puede iterar sobre un diccionario, pero de esta forma sólo se van a
extraer
#las llaves (que en teoría, debido a que los pares llave-valor de los
#diccionarios no tienen un orden entre ellos, podrían extraerse/imprimirse en
#cualquier orden arbitrario, no necesariamente en el orden en el que se
#ingresaron al crear el diccionario, ya que los diccionarios son como
conjuntos):
print("Iteración sobre (los valores clave/llave de) un diccionario:")
res=""
for x in diccionario:
    res=res+str(x)+" "
print(res)
```

```
#Se puede hacer de esta manera para ver el diccionario completo (aunque cada
#par llave-valor lo presenta Python como una tupla):
print("Iteración sobre el diccionario completo (pares clave-valor):")
res=""
for x in diccionario.items():
    res=res+str(x)+" "
print(res)

#Si se deseara extraer sólo los valores del diccionario, se puede:
print("Iteración sobre los valores de un diccionario, versión 1:")
res=""
for ind,val in diccionario.items():
    res=res+str(val)+" "
print(res)

#Muchos acostumbran hacer lo anterior de la sig. manera (usando la "variable
#anónima" de Prolog como primer elemento de cada tupla, ya que no se va a usar
#más adelante), pero en realidad "_" es un nombre de variable válido en
#Python, y ocupa memoria, como se puede ver a través del "print" que se
incluyó
#dentro del "for", así es que realmente es equivalente a la versión anterior
#(en donde se usó "ind" en lugar de "_" para el primer elemento de cada
tupla):
print("Iteración sobre los valores de un diccionario, versión 2:")
res=""
for _,val in diccionario.items():
    print(_)
    res=res+str(val)+" "
print(res)

#Si no se desea iterar por completo sobre una colección (que es lo que hace
#la instrucción "for"), se puede iterar parcialmente obteniendo un objeto
#iterador (que permite iterar sobre la colección de interés), usando la
función
#"next" explícitamente cada vez que uno quiere moverse al siguiente dato de la
#colección, y usando "while" para que las repeticiones se puedan detener antes
#de terminar de iterar por completo (en caso de que así se desee).  Aquí se
#ejemplifica con una lista, pero se puede hacer lo mismo con cualquier tipo de
#colección:
print("Iteración parcial sobre los elementos de una lista:")
#Obtiene un objeto que sirva para iterar sobre la lista
it=iter(lista)
#Regresa el siguiente dato dentro de la colección sobre la cual se está
#iterando (ojo: el argumento no es la colección, sino un objeto iterador
#asociado con la colección):
elemento=next(it)
res=""
while(elemento!="d"):
    res=res+str(elemento)+" "
    elemento=next(it)
print(res)
```

```
#Otras dos versiones, esta vez llegando a iterar sobre toda la colección, lo
#cual hace que se lance una excepción de tipo StopIteration cuando se intente
#usar "next" en caso de que no haya un siguiente elemento de la colección en
#ese momento.  Los "try-except" sirven para manejar la excepción y hacer que
#el programa siga ejecutándose cuando ocurre, en lugar de "tronar" en ese
#momento, así robusteciendo el programa:
print("Iteración completa con 'while' sobre los elementos de una lista:")
it=iter(lista)
elemento=next(it)  #Aquí sabemos que va a haber por lo menos un dato
res=""
try:
    while(elemento!="z"):
        res=res+str(elemento)+" "
        elemento=next(it)  #Aquí es donde podría ocurrir la excepción
except StopIteration:
    print("Se acabó la colección")
print(res)

#Aquí la lista está vacía, por lo que el primer "next" (antes del ciclo) es
#el que debe lanzar la excepción, no el "next" que se tiene dentro del ciclo:
print("Iteración con 'while' sobre los elementos de una lista vacía:")
lis=[]
it=iter(lis)
try:
    elemento=next(it)  #Aquí debe lanzarse la excepción porque no hay datos.
    #No se debe ejecutar el resto de lo que hay dentro del "try", debido a que
    #debe haber ocurrido una excepción en la instrucción anterior, por lo que
    #el flujo de la ejecución debe pasar al "except" que corresponda a la
    #excepción ocurrida (o a un "except" general):
    res=""
    while(elemento!="d"):
        res=res+str(elemento)+" "
        elemento=next(it)
        print(res)
except StopIteration:
    print("No hay elementos en la lista sobre los que se pueda iterar.")
```

Un ejemplo con la función predefinida "type":
Debido a la heterogeneidad de muchas de las colecciones de Python, así como el hecho de que no es necesario declarar los tipos de las variables, muchas veces es necesario en Python integrar a los programas la detección de los tipos de los datos. Esto se logra a través del uso de la función predefinida `type`. Aquí se muestra un ejemplo de su uso:

```
#En este ejemplo se crea una lista heterogénea que contiene un número entero,
#un número real, una cadena de caracteres, un valor booleano, una tupla y un
#tipo de datos:
lista=[1,3.5,"what?",True,(1,"bla"),bool]
print(lista)
#Dentro del ciclo se procesan los elementos de la lista, se detecta su tipo,
#y dependiendo del tipo se imprime un mensaje distinto.  La detección del
#tipo se logra a través de la función predefinida "type" y se compara el
#valor obtenido (que es un objeto de tipo "type", no es una cadena de
#caracteres ni nada por el estilo) con cada uno de cinco opciones previstas
#(además de contar con un "else" para cualquier otra opción):
for i in lista:
    t=type(i)
    if t==int:
        print("Es un número entero.")
    elif t==float:
        print("Es un número real.")
    elif t==str:
        print("Es una cadena de caracteres.")
    elif t==bool:
        print("Es un valor booleano.")
    elif t==type:
        print("Es un tipo de datos.")
    else:
        print("Es algún otro tipo de valor.")
```

Algunos paquetes útiles que existen en Python:

- Para la generación de números aleatorios: `random`.
- Para consultar el reloj del procesador: `time`.
- Para graficar datos numéricos: `matplotlib.pyplot`.
- Para el manejo de arreglos (más eficientes que las listas porque los arreglos son heterogéneos): `numpy`.
- Para cálculos científicos, matemáticos e ingenieriles: `scipy`.
- Para la manipulación y el análisis de tablas de datos numéricos: `pandas`.

Dos ejemplos de la sintaxis utilizada para importar paquetes como éstos:

```
import random
import matplotlib.pyplot as plt
```

A partir de la importación se pueden utilizar los métodos disponibles en los paquetes, por ejemplo:

```
x=random.randint(0,999)
plt.plot(…)
```