Strings e I/O

Los strings

En la clase anterior vimos que existen los **strings** en Python, también llamados **cadenas**, es un tipo de dato que sirve para almacenar texto y que internamente no es más que una **lista de caracteres inmutables** que se representan como una secuencia de caracteres encerrados entre comillas simples o dobles (“Hola mundo” u ‘Hola mundo’ son strings). Como hemos visto, este tipo de dato es muy útil, pues nos permite almacenar palabras y, de este modo, crear programas que se comunican con el usuario, entre otras funcionalidades que veremos más adelante.

Los strings, al igual que todos los tipos de datos que hemos revisado hasta el momento (`int`, `float`, `long`, `boolean` y `list`) tienen operaciones y funciones nativas que permiten operar con ellas. Por ejemplo:

- Concatenación: `<string> + <string>`, al igual que el operador concatenación de las listas, une dos cadenas en una sola.
- Repetición: `<string> * <int>`, al igual que el operador de repetición de las listas, repite una cadena varias veces.
- Largo: `len(<string>)`, devuelve la cantidad de elementos de una cadena.

Al igual que en las listas, los strings usan una codificación posicional para acceder a sus elementos, así para un string `saludo = 'HOLA MUNDO'` (figura 1) tenemos que podemos acceder al carácter en la posición 0 utilizando `saludo[0]`, al segundo como `saludo[1]` y así sucesivamente.

0	1	2	3	4	5	6	7	8	9
H	O	L	A		M	U	N	D	O

Figura 1. Manejo interno del string “Hola Mundo”

Sin embargo, ¿qué sucede si utilizamos un valor negativo?, por ejemplo `saludo[-2]`.

Ejemplo 1

```
>>> saludo = 'HOLA MUNDO'
>>> saludo[0]
'H'
>>> saludo[-2]
'D'
>>>
```

Sucede que nos entrega el penúltimo valor, es decir, con números negativos ¡podemos acceder a elementos desde el final hacia el inicio del string!

Pero en Python, un string es un **objeto**. En palabras muy sencillas, esto quiere decir que el tipo de dato también tiene un **comportamiento** que le es característico y que está definido por un conjunto de **métodos**. Así, podemos pedir a un string que "haga algo" invocando alguno de sus métodos. Por ejemplo:

- `<string>.lower()` solicita al string que devuelva un nuevo string con el mismo contenido pero asegurando que todas las letras están en minúsculas.
- `<string>.upper()` solicita al string que devuelva un nuevo string con el mismo contenido pero asegurando que todas las letras están en mayúsculas.

Ejemplo 2

```
>>> saludo.lower()
'hola mundo'
>>>
>>> 'chao mundo cruel!'.upper()
'CHAO MUNDO CRUEL!'
>>>
```

Pregunta 1

Con tu grupo, trabajen en la primera pregunta de la actividad.

Entrada en Python

Hasta el momento hemos visto la comunicación entre Python y el usuario a través de llamadas a **funciones**, evaluación de **operaciones matemáticas y booleanas** y las sentencia `print`. Esta última es vital para que un programa sea útil, ya que permite entregar a un usuario los resultados que obtiene un programa en Python, es decir, permite que el computador entregue **respuestas** al usuario.

Las funciones reciben entradas del usuario a través de sus **argumentos**, sin embargo, muchas veces estos parámetros no los conocemos *a priori* y, por esta razón, los lenguajes de programación proveen **sentencias de entrada** que permiten **alimentar** un programa con valores en tiempo de ejecución. En Python, si bien existen **dos** sentencias de entrada principales, hasta ahora hemos trabajamos con la sentencia `input()` que

permite el ingreso de una **expresión** en Python. Debemos recordar que una **expresión** es la combinación de valores, variables y símbolos reservados, que es interpretada de acuerdo a las reglas particulares del lenguaje Python. Es decir, la entrada que ingrese un usuario **debe respetar** las normas **léxicas**, **sintácticas** y **semánticas** que el lenguaje impone. De lo contrario, se produce una situación no deseada.

Qué entra, qué sale

Teniendo una sintaxis y una semántica que permite el ingreso y salida de datos no indica cuándo debe usarse estas sentencias. Lo que determina con qué datos se debe alimentar un programa y qué datos debe entregar un programa es **el problema que resuelve**.

Pero un buen ingeniero también debe pensar en que su solución sea útil en la mayor cantidad de situaciones posible, es decir pensar en **soluciones más generales** que lo que se requiere para una situación específica. Esta **actitud** puede traer ahorro de tiempo y recursos en el futuro, al poder **reutilizar** el programa, convirtiendo una solución informática en una herramienta **eficiente**.

Por esta razón es que tratamos de programar funciones genéricas y hacemos el ingreso de datos separadamente para el problema que se está resolviendo. Si bien esto puede parecer más engorroso que simplemente programar llamadas a funciones específicas, en largo plazo ahorraremos trabajo y nuestra solución es de mejor calidad.

Los mismos criterios deben ser considerados cuando se decide qué es lo que un programa ha de entregar como respuesta. Esto obedece a que el usuario de la solución es un **ser humano** y por lo tanto la solución informática genera una *interacción humano-computador*. Entre muchas otras cosas, este criterio sugiere tener siempre en consideración que usar una herramienta informática ha de ser una tarea lo más intuitiva posible, que no lleve al usuario a hacer interpretaciones erróneas de lo que el software pide o entrega. Por ahora, esto va a significar que un programa debe entregar **mensajes informativos** al usuario cuando se realice entrada o salida de datos y realizar el ingreso de datos de la forma que sea más fácil para el **usuario**, y no para el **programador**.

Sin embargo, este criterio no siempre manda puesto que, muchas funciones se definen únicamente para cálculo interno, como por ejemplo las funciones auxiliares que se definen para simplificar un cálculo matemático complejo. Dichas funciones no entregan mensajes al usuario por pantalla, pues se definen de acuerdo a los criterios de **generalidad** y **reusabilidad** mencionados arriba.

Esta restricción impide que usemos la sentencia `input()` para recibir cualquier cosa, debido a que en algunos casos Python no sabrá cómo interpretar un dato.

Ejemplo 3

```
>>> valor = input("Ingrese texto: ")
Ingrese texto: fundamentos

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    valor = input("Ingrese texto: ")
  File "<string>", line 1, in <module>
NameError: name 'fundamentos' is not defined
>>>
```

Podemos ver que el intérprete de Python informa de un error al ingresar un texto sin comillas, pues considera dicho dato como una variable a la que se está referenciando. Un usuario inexperto que desconoce el funcionamiento de Python podría perfectamente ingresar texto sin comillas, y el programa fallaría rápidamente.

Para evitar esta situación es común utilizar la sentencia `raw_input()`, que automáticamente **convierte lo que el usuario escribe en un string**, asegurando que todas entrada sea válida en Python, ya que a diferencia de `input()`, esta sentencia no aplica las reglas sintácticas y semánticas de la evaluación de expresiones.

Ejemplo 4

```
>>> valor = raw_input("Ingrese texto: ")
Ingrese texto: fundamentos 1+2+3
>>>
>>> valor
'fundamentos 1+2+3'
>>>
```

Pregunta 2

Con el grupo respondan la pregunta 2 de la actividad de hoy

Manejo de archivos

Hasta ahora hemos visto cómo leer datos desde **teclado**. Si bien esto ayuda a construir programas útiles, es bastante común que se utilicen otras **fuentes** para los datos de entrada, especialmente **archivos** (también llamados **ficheros**). Un **archivo** se identifica

por una **ruta** (en el árbol de directorios del computador) y un **nombre**. Es necesario conocer esta identificación para poder acceder a un archivo. Por simplicidad, en la siguiente discusión se supondrá que el archivo se encuentra en el **mismo directorio** que el programa que lo utiliza. Para acceder al contenido de un archivo, primero éste debe ser **abierto** con una sentencia **open**, que tiene la siguiente sintaxis:

Importante

```
<identificador> = open ('<archivo>'[, '<modo>'])
```

Donde el modo puede ser **r** (reading), **w** (writing), **a** (append)

Los modos indican la forma en que el programa utilizará el archivo:

- El modo de **lectura** es indicado con la letra `r` (del inglés **reading**). En este modo, un programa sólo puede leer el contenido del archivo, pero éste no puede ser modificado. Por esta razón, el archivo debe **existir** en el directorio. Este es el modo que asume Python si el modo se omite en la sentencia `open`.
- El modo de **escritura** es indicado con la letra `w` (del inglés **writing**). En este modo, un programa sólo puede escribir contenido en el archivo. Si el archivo no existe, éste **se crea**. Si el archivo ya existe, su contenido se **sobrescribe** y se pierde el contenido original.
- El modo de **añadidura** es indicado con la letra `a` (del inglés **append**). En este modo, un programa sólo puede escribir contenido en el archivo, añadiéndolo al final sin sobrescribir el contenido original que tiene al momento de la apertura. Si el **archivo** no existe, éste también es creado por este modo de acceso.

En todos los casos cuando realizamos un acceso inválido (como intentar leer un archivo abierto en modo escritura o escribir en un archivo abierto para lectura), el intérprete de Python gatilla un **error de entrada/salida**. El siguiente ejemplo muestra lo que ocurre cuando intentamos abrir en modo lectura un archivo que no existe.

Ejemplo 5

```
>>> archivo = open('casa.txt','r')

Traceback (most recent call last):
  File "<pyshell#54>", line 1, in <module>
    archivo = open('casa.txt','r')
IOError: [Errno 2] No such file or directory: 'casa.txt'
```

Si analizamos la sintaxis de la sentencia open, podremos ver que incluye una sentencia de asignación. Esto porque la sentencia **crea y devuelve un objeto** en memoria que mantiene la información necesaria para acceder al contenido del archivo. Para utilizar este objeto, le asociamos un identificador a través de la asignación. Este objeto es de tipo **archivo** (`file` en Python) y es otro tipo de dato nativo manejado por el lenguaje. Los **métodos** definidos para este tipo de objetos nos permiten realizar acciones sobre el archivo que representan. Entre los métodos más importantes que debemos conocer están:

- `<file>.close()` que indica que el archivo administrado debe **cerrarse**. Esto es necesario para que el archivo se almacene en el dispositivo de almacenamiento del equipo, es decir, enviar los datos desde **memoria principal**, donde están siendo operados y consultados, al **disco duro** o unidad de datos donde son almacenados de forma permanente.
- `<file>.readline()` que **lee la siguiente línea en un archivo abierto** y que **devuelve como un string**. Notemos que una **llamada** al método `.readline()` transfiere el contenido de la línea actual del archivo y **avanza** a la siguiente línea. En este sentido, un archivo puede leerse solamente en forma **secuencial** y el programa no puede devolverse a líneas anteriores. También debemos notar que una llamada a `.readline()` sólo es válida cuando un archivo permanece abierto en modo de lectura.
- `<file>.readlines()` que **lee todas las líneas del archivo** de una sola vez, y las **devuelve como una lista de strings** (cada elemento es una línea). Notemos que este método es útil para archivos de tamaño reducido pero es inconveniente para archivos grandes por su un alto consumo de memoria.
- `<file>.write(•)` que **escribe el contenido de un string (argumento) en el archivo**. Es importante mencionar que la función **no escribe líneas** y que, para hacerlo, es necesario agregar el **carácter fin de línea** (`'\n'`) al final del string que se está escribiendo para se pase a la línea siguiente del archivo. También debemos notar que una llamada a `.write(•)` sólo es válida cuando un archivo permanece abierto en modo de escritura o de añadidura.

Probemos el código `escribir.py`, guardando el programa en un directorio conocido y revisemos qué sucede.

Podremos ver que, en el directorio donde guardamos el programa, encontramos un archivo llamado “escribir.txt” con las mismas líneas que ingresamos al ejecutar el programa.

escribir.py

```
# -*- coding: cp1252 -*-

#
# Funciones
#

#
# Función que escribe líneas de texto en un archivo
# Entrada:
# Salida:
# Asegura: se ha creado el archivo 'escribir.txt' con el texto
#          ingresado, línea a línea, por el usuario.
#
def escribir():

    # Se abre el archivo, en modo de escritura
    archivo = open('escribir.txt', 'w')

    # Mientras el usuario no indique que quiere salir del procesamiento
    salir = False
    while not salir:
        # Se lee un string a escribir
        linea = raw_input("Ingrese línea a escribir, 'exit' para salir: \n")
        # Si el usuario escribió la palabra 'exit',
        # se debe salir del ciclo.
        # Sino, se concatena la línea leída con un salto de línea
        # y se escribe en el archivo.
        if linea == 'exit':
            salir = True
        else:
            linea = linea + '\n'
            archivo.write(linea)

    # Se guardan los cambios en disco cerrando el archivo
    archivo.close()

    # Se informa de que el archivo se ha creado correctamente
    print("Archivo creado correctamente")



#
# Bloque principal
#

escribir()
```

Veamos un ejemplo completo: ejecutemos el programa `escribir.py` y escribamos cinco líneas; luego podremos consultarlas directamente en el intérprete de Python abriendo nuevamente el archivo (recordemos que el programa terminaba cerrando el archivo), y utilizando el método `.readline()` para ir leyéndolo.

Ejemplo 6

```
>>> ============================ RESTART ============================
>>>
Ingrese línea a escribir, 'exit' para salir:
hola a todos
Ingrese línea a escribir, 'exit' para salir:
cómo están
Ingrese línea a escribir, 'exit' para salir:
espero que bien
Ingrese línea a escribir, 'exit' para salir:
chauu!
Ingrese línea a escribir, 'exit' para salir:
exit
Archivo creado correctamente
>>>
>>> archivo = open('escribir.txt', 'r')
>>> print archivo.readline()
hola a todos

>>> print archivo.readline()
cómo están

>>> print archivo.readline()
espero que bien

>>> print archivo.readline()
chauu!

>>> print archivo.readline()

>>> print archivo.readline() == ''
True
>>>
```

De este modo podemos ver que Python avanza en el archivo **una línea cada vez hasta llegar al final del archivo**. Notemos que el string devuelto para cada línea incluye el carácter `'\n'` encontrado al final. También notemos que cuando las líneas se acaban, el método `.readline()` devuelve un string vacío.

El programa `leer.py` crea una función que usa esta lectura línea a línea, la que podemos usar para mostrar por pantalla el contenido del archivo.

Notemos que en este caso, no estamos utilizando *explícitamente* el método `.readline()`, sino que un **ciclo `for-in`** que recorre el archivo línea a línea. De este modo Python realiza la operación `linea = archivo.readline()` en cada paso del ciclo de forma **implícita**, haciendo la interfaz con archivos bastante natural. También notemos que hemos usado una coma al final de la sentencia que muestra el string de una línea por pantalla, ya que éste incluye el fin de línea.

leer.py

```
# -*- coding: cp1252 -*-

#
# Funciones
#

#
# Función que lee líneas de texto en un archivo
# Entrada:
# Salida:
# Requiere: que el archivo "escribir.txt" exista
# Asegura: se ha mostrado en pantalla, línea a línea,
#          el contenido del archivo de texto 'escribir.txt'.
#
def leer():

    # Se abre el archivo, en modo de lectura
    archivo = open('escribir.txt', 'r')

    # Para cada línea en el archivo
    for linea in archivo:
        # Se muestra en pantalla
        print linea,

    # Se cierra el acceso al archivo
    archivo.close()

    # Se informa de que el archivo se ha leído correctamente
    print("-- Archivo leído correctamente--")



#
# Bloque principal
#

leer()
```

Es importante señalar que guardar valores en un archivo, no sólo sirve para luego realizar funciones como el `leer()`, sino que es posible combinar la lectura de archivos con funciones de manejo de strings y listas para crear programas que realicen cálculos mucho más complejos. Adjunto a este apunte encontrarás un programa de ejemplo más complejo para que revises más tarde.

Salida con formato

Otra necesidad a la hora de comunicarse con el usuario, tiene que ver con el **formato de la salida** de los datos que entrega un programa. Por ejemplo, puede que se requiera colocar datos con diferentes valores ordenados en columnas, como se muestra en el ejemplo 7.

Ejemplo 7

```
>>>
     MARIO CERECEDA    100 25% 025.00
 ALEJANDRA SOTO       1500 05% 075.00
      JUAN MCSILVA       5 08% 004.00
>>>
```

Para poder escribir un dato con formato, Python utiliza strings especiales llamados **strings formato**, que contienen **marcadores** que reservan un lugar dentro del string para colocar datos. Existen básicamente tres tipos de marcadores: `%s` para una cadena de texto (string), `%i` (o `%d`) para un número entero y `%f` para un valor flotante.

Por ejemplo, el string formato `'%d %s %d es igual a %f'` indica que se ha de reservar un lugar para un valor entero, seguido de un lugar para un string, seguido de un lugar para otro valor entero, seguido del texto "`es igual a`" y que finaliza con un lugar reservado para un valor flotante (todos estos elementos separados con un espacio).

Para entregar los valores que deben colocarse en las posiciones determinadas por los marcadores se utiliza el operador **%** que opera un string formato con una **lista de valores** delimitados en **paréntesis redondos**. El primer marcador asume el primer valor de la lista, el segundo marcador asume el segundo valor de la lista y así sucesivamente. Por esta razón, el número de valores indicados en la lista debe **coincidir** con el número de marcadores contenidos por el string formato, como puede verse en el ejemplo 8.

Importante

Note que éste el uso del operador % para la salida con formato corresponde a una utilización totalmente distinta a la **operación resto** entre números enteros. La utilización de un símbolo, función o variable, con **más de un uso**, se denomina **sobrecarga** y es común en muchos lenguajes de programación.

Ejemplo 8

```
>>> '%d %s %d es igual a %f' % (3, '*', 3)

Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    '%d %s %d es igual a %f' % (3, '+', 3)
TypeError: not enough arguments for format string

>>> '%d %s %d es igual a %f' % (3, '*', 3, 3 * 3)
'3 * 3 es igual a 9.000000'
>>>
```

Puede apreciarse que los valores entregados en la lista pueden perfectamente ser **expresiones**, las que se evalúan al instante de ser posicionadas en el string formato. También es importante notar que si un valor **no corresponde** al tipo de dato indicado por su respectivo marcador, Python intenta realizar la **conversión** desde el tipo original al requerido. Por esta razón, en el ejemplo anterior se muestra el valor flotante `9.000000` a pesar que la expresión `3 * 3` genera un valor entero. Cuando la conversión no es posible, se gatilla un **error de tipo**.

Para poder otorgar formatos más complejos a los datos existen **modificadores** que afectan a los diferentes **marcadores**:

`%<n>s` `%<n>i` (o `%<n>d`) `%<n>f`

Un número entero positivo `n` después del caracter `%` indica que el dato debe escribirse con **a lo menos** `n` caracteres. Si el valor especificado para el marcador se representa normalmente con menos caracteres, entonces se agregan espacios a la **izquierda** para completar este tamaño.

`%<-n>s` `%<-n>i` (o `%<-n>d`) `%<-n>f`

Un número entero negativo `n` después del caracter `%` indica que el dato debe escribirse con **a lo menos** `n` caracteres, pero en este caso si se requiere agregar espacios para completar este tamaño, éstos se agregan a la **derecha** de la representación usual del valor.

`%0<n>i` (o `%0<n>d`) `%0<n>f`

Especificando un número entero positivo que comienza con un cero después del caracter `%` indica que el dato numérico debe escribirse con **a lo menos** `n` caracteres, y que cuando la representación usual del valor sea menos caracteres debe completarse este tamaño con **ceros a la izquierda**.

`%.<p>f`

Agregando un punto y luego un número entero positivo `p` después del caracter `%` indica que el dato flotante debe escribirse con `p` dígitos decimales. Cuando el flotante tiene más de `p` dígitos decimales, éste se redondea al **p-avo** dígito

decimal. Si el flotante no tiene `p` dígitos decimales, se escriben ceros a la derecha hasta completar los `p` dígitos.

`%<n>.<p>f %0<n>.<p>f`

Al mezclar los **modificadores** en un **marcador** de flotante debe tenerse claro que el punto decimal y el número de dígitos decimales se cuentan como **parte del tamaño total**, es decir `n` debiera ser mayor que `p` + 1 en estos casos.

De esta forma, el código que muestra la salida con formato del ejemplo 7, se muestra en el programa `tabla.py`. Notemos que para poder escribir el símbolo de porcentaje debemos usar el marcador especial `%%`.

tabla.py

```
>>> def Ejemplo7():
    print '%10s %-10s %4i %02i%% %06.2f' % ('MARIO', 'CERECEDA', 100, 25, 100*25/100.0)
    print '%10s %-10s %4i %02i%% %06.2f' % ('ALEJANDRA', 'SOTO', 1500, 5, 1500*5/100.0)
    print '%10s %-10s %4i %02i%% %06.2f' % ('JUAN', 'MCSILVA', 5, 8, 50*8/100.0)
```

Pregunta 3

Con el grupo respondan la pregunta 3 de la actividad de hoy

TAREA

1. **(Muy Fácil)** Estudiar el ejemplo adjunto a este apunte y el resumen de funciones en Python.
2. **(Fácil)** Modifica el ejemplo para que se generen 100 valores aleatorios entre 1 y 100, con ello se calcule el mínimo, el máximo y el promedio de la serie de valores, y que todas estas operaciones se realicen en un solo programa.
3. **(Medio)** Además de las operaciones descritas en (2), crea una función que ordene los 100 valores de menor a mayor y los escriba en un archivo "salida.txt", no puedes usar la función `sort()`
4. **(Difícil)** Resuelve (3) sin mirar libros, internet o cualquier otro recurso.
5. **(Muy Difícil)** Añada al programa original tres funciones:
   - La función descrita en (3).
   - Una función que calcule la desviación estándar con 3 decimales.
   - Una función que calcule la moda de la muestra.