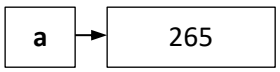
Ficha 3

Tipos Estructurados Básicos

1.] Tipos estructurados (o compuestos) básicos en Python: secuencias de datos.

Hemos visto que en Python una variable puede ser asignada con un valor y luego cambiar ese valor por otro (perdiendo el valor anterior) cuando el programador lo requiera. En general, aquellos tipos de datos que sólo admiten que una variable pueda contener *un único valor de ese tipo*, se llaman *tipos simples*. Los tipos *int*, *float* y *bool* que ya conocemos, son ejemplos de tipos simples. Una variable de tipo simple, entonces, puede representarse como un recipiente con un único compartimiento para almacenar un único elemento:

```
a = 265
print(a)
```



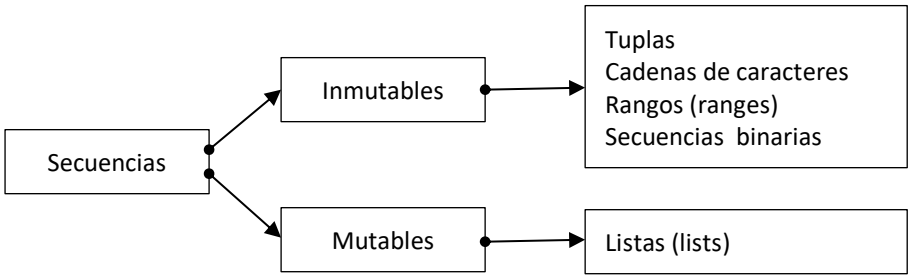
Sin embargo, tanto en Python como en todo otro lenguaje, existen tipos de datos que permiten que *una misma variable pueda contener varios valores al mismo tiempo*. Estos tipos se designan genéricamente como *tipos compuestos* o *tipos estructurados* y por esta razón, una *variable de tipo estructurado* se suele designar también como una *estructura de datos*.

Hemos visto ya un ejemplo concreto de tipo estructurado: las *cadenas de caracteres* (o *strings*), que son casos particulares de una gran familia de tipos compuestos de Python llamados *secuencias* (o *secuencias de datos*), las que a su vez pueden ser *inmutables* o pueden ser *mutables*.

Una *secuencia de datos inmutable* es simplemente un conjunto finito de datos cuyos valores y orden de aparición ***no pueden modificarse una vez asignados***. Las ya citadas *cadenas de caracteres* son un ejemplo de *secuencias inmutables*.

Por otra parte, las *secuencias de datos mutables* son aquellas secuencias en las que los valores individuales ***pueden modificarse*** luego de ser asignados. La *Figura 1* muestra un esquema general de los tipos de secuencias disponibles en *Python 3* (aunque considere que este esquema podría ampliarse en el futuro, ya que Python continuamente incorpora mejoras y agregados en sus nuevas versiones [1]):

Figura 1: Tipos de *secuencias* en Python 3.



Hasta aquí sólo hemos visto ejemplos muy simples de uso de *cadenas de caracteres*, pero el resto de los tipos de secuencias de la figura anterior irán siendo presentadas y estudiadas a lo largo del curso.

A diferencia de las variables de tipo simple, se puede pensar que una *estructura de datos* (o sea, una *variable de tipo compuesto*) es un recipiente que contiene muchos compartimientos internos.

Así, una *cadena de caracteres* puede entenderse como un contenedor dividido en tantas casillas como caracteres tenga esa cadena. Y si el programador necesita almacenar varios números (o varios elementos de cualquier tipo que no sean necesariamente caracteres) en el mismo contenedor, puede en Python usar *tuplas* (entre otras alternativas).

Una *tupla* es una *secuencia inmutable* de datos que pueden ser de tipos diferentes. En Python hay varias formas de definir una tupla *t* que contenga ninguno, uno o varios valores iniciales [1]:

1. Usando un par de paréntesis vacíos para denotar una tupla vacía: `t = ()`
2. Usando una coma al final, para denotar una tupla de un solo valor: `t = a,` o bien `t = (a,)`
3. Separando los ítems con comas: `t = a, b, c` o bien `t = (a, b, c)`
4. Usando la función predefinida `tuple()`: `t = tuple()` o bien `t = tuple(iterable)` (donde *iterable* es cualquier secuencia que pueda recorrerse con un *ciclo for iterador*[1]: una cadena, otra tupla, un rango, una lista, etc. Por ejemplo, la siguiente instrucción crea una tupla *t* con cada uno de los caracteres de la cadena 'Mundo': `t = tuple('Mundo')`).

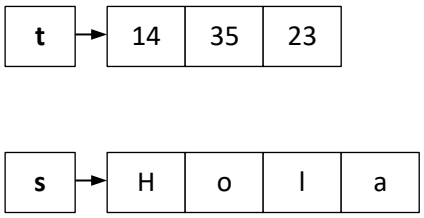
Note que el uso de paréntesis para encerrar una tupla es opcional (ver forma 3), aunque a veces es exigible (forma 1) para evitar ambigüedades.

Se dice que la tupla *t* del script que sigue tiene tres elementos o componentes (uno por cada número), y la cadena *s* tiene cuatro (uno por cada caracter de la cadena). Los esquemas gráficos que acompañan al script ilustran la idea:

Figura 2: Ejemplos gráficos de secuencias inmutables de tipo *cadena* y *tupla*:

```
# una tupla…
t = 14, 35, 23
print(t)

# una cadena de caracteres…
s = 'Hola'
print(s)
```



Cualquiera sea el tipo de secuencia que se esté usando (tupla, cadena, lista, etc.) la función interna *len()* de Python puede usarse para determinar la *cantidad de elementos que tiene esa secuencia*:

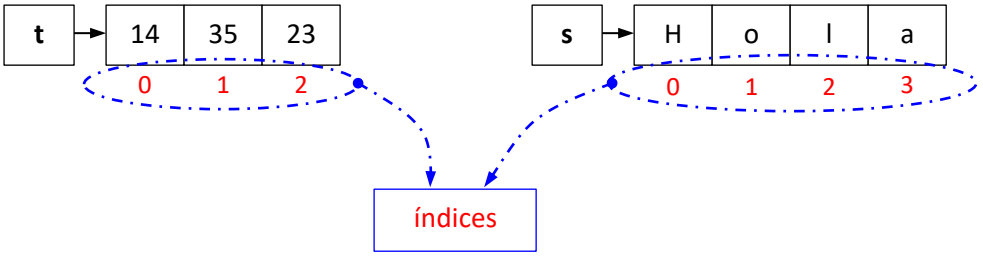
```
t = 14, 35, 23
n = len(t)
print(n)      # muestra: 3
```

[1] Veremos el concepto de *ciclo for iterador* en una ficha posterior, en ocasión de estudiar el funcionamiento de una *instrucción repetitiva*.

```
s = 'Hola'
m = len(s)
print(m)      # muestra: 4
```

Un detalle importante que por ahora sólo veremos a modo de presentación (y sobre el cual volveremos en profundidad más adelante en el curso) es que cada casillero de una secuencia de cualquier tipo está asociado a un número de orden llamado *índice*, mediante el cual se puede acceder a ese casillero en forma individual. En Python, la *secuencia de índices* que identifica a los casilleros de una secuencia de datos comienza desde *0(cero)* y sigue luego en forma correlativa. La *Figura 3* ayuda a entender la idea (suponga la tupla *t* y la cadena *s* del ejemplo anterior):

Figura 3: Dos secuencias y sus casilleros identificados por índices.



Para acceder a un elemento individual de una secuencia, basta con escribir *el identificador de la misma y luego agregar entre corchetes el índice del casillero a acceder*:

```
t = 14, 35, 23
print(t[1])      # muestra: 35
x = t[2]
print(x)         # muestra: 23

s = 'Hola'
print(s[0])      # muestra: H
c = s[3]
print(c)         # muestra: a
```

Cuando se trabaja con secuencias de tipo ***inmutable*** (ver *Figura 1*) como las tuplas o las cadenas de caracteres, *cualquier intento de* ***modificar*** *el valor de un casillero individual accediéndolo por su índice producirá un error de intérprete*, similar a los que se muestran aquí:

```
t = 14, 35, 23
t[2] = 87      # error…
TypeError: 'tuple' object does not support item assignment
```

```
s = 'Hola'
s[2] = 'a'      # error…
TypeError: 'str' object does not support item assignment
```

La modificación del valor de un elemento de una secuencia está permitida siempre y cuando esa secuencia sea del tipo *mutable*, como es el caso de las *listas* en Python que veremos oportunamente [2].

Es perfectamente válido que una *tupla* contenga elementos de tipos diferentes: cada casillero se define en el momento que se asigna, y dado que Python es de tipado dinámico, nada impide que esos casilleros sean de diferentes tipos:

```
t2 = 'Juan', 24, 3400.57, 'Córdoba'
print(t2)    # muestra: ('Juan', 24, 3400.57, 'Córdoba')
```

Si se desea procesar por separado cada elemento, se usan los índices como se indicó y se pueden introducir variables temporales a criterio y necesidad del programador:

```
nombre = t2[0]
edad = t2[1]
print('Nombre:', nombre, '- Edad:', edad, '- Sueldo:', t2[2], '- Ciudad:', t2[3])
# muestra:    Nombre: Juan - Edad: 24 - Sueldo: 3400.57 - Ciudad: Córdoba
```

Habiendo introducido el uso de *tuplas* es muy interesante observar que en Python una *expresión de asignación* puede estar formada por *tuplas en ambos lados de la expresión*: una secuencia de variables (separadas por comas) en el lado izquierdo del operador de asignación, y otra secuencia con la misma de cantidad de valores, variables y/o expresiones del lado derecho del operador de asignación, también separados por comas. En el siguiente ejemplo sencillo, las variables *x* e *y* forman una tupla en el lado izquierdo, y toman respectivamente los valores 10 y 20 (que surgen de las dos expresiones que forman otra tupla en el lado derecho de la asignación):

```
a = 5
x, y = 2*a, 4*a
```

En la expresión de asignación el miembro derecho es evaluado primero, antes que el miembro izquierdo, y la evaluación procede de izquierda a derecha. Los valores obtenidos de la *tupla* derecha son provisoriamente asignados en variables temporales y luego esos valores son asignados a las variables de la *tupla* del lado izquierdo, también en orden de izquierda a derecha (por este motivo el 10 se asignó en *x* y no en *y*: el primer valor del lado derecho, se asigna en la primera variable del lado izquierdo, y así sucesivamente). La gráfica que se muestra en la *Figura 4* (*página 56*) muestra la forma en que todo ocurre.

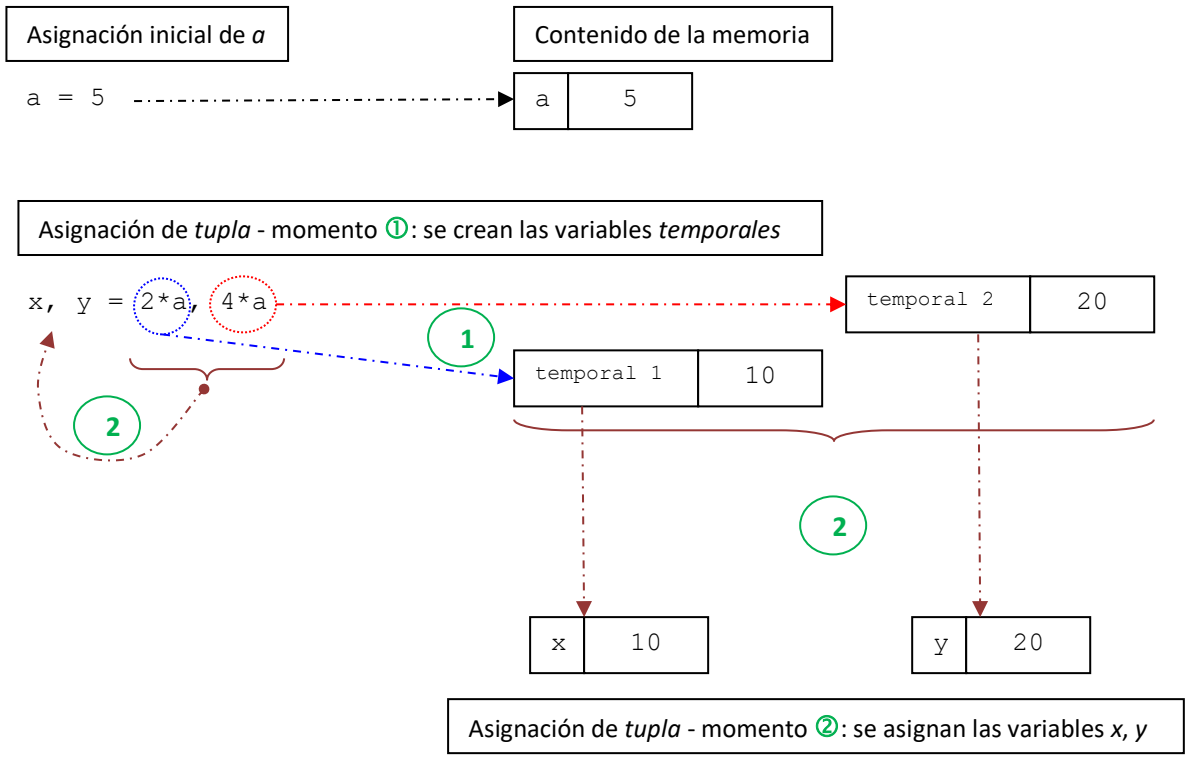
En una expresión de asignación de múltiples operandos (como la de una *tupla*), el operador *coma* (,) actúa como un *empaquetador* cuando está a la derecha de la asignación, produciendo la *tupla* (o *secuencia de expresiones*) con los valores obtenidos. Y actúa como un *desempaquetador* si está a la izquierda de la expresión [1]. La siguiente instrucción es válida en Python, y crea una *tupla* de dos números (5 y 8) almacenada como tal en la variable *sec*:

```
sec = 5, 8
print('Secuencia: ', sec)
# muestra: Secuencia: (5, 8)
```

Y como era de esperarse, la instrucción siguiente asigna los valores 5 y 8 en las variables *a* y *b*:

```
a, b = sec
print('a: ', a)
print('b: ', b)
```

Figura 4: Asignación de una tupla en Python.



Lo anterior tiene al menos una aplicación práctica inmediata: es muy común que el programador tenga que realizar lo que se conoce como un *intercambio de valores entre dos variables*: se tienen dos variables *a* y *b*, cada una con un valor previo, y se necesita intercambiar esos valores, de forma que *a* tome el valor de *b*, y que *b* tome el valor original de *a*. Según sabemos, sería un error intentar simplemente lo siguiente:

```
a = 12
b = 4

a = b
b = a

print('Valor final de a: ', a)
print('Valor final de b: ', b)
```

ya que la secuencia mostrada hará que ambas variables queden valiendo finalmente el valor 4, perdiendo el 12. Asegúrese de entender la veracidad de esta afirmación, haciendo un gráfico de los valores de cada variable en cada momento del esquema.

¿Cómo hacer entonces el intercambio? En general, en la mayoría de los lenguajes de programación, el intercambio de valores de dos variables requiere el uso de una tercera variable a modo de auxiliar, y usar (obviamente…) ingenio y sentido común. La siguiente secuencia de instrucciones en Python produce correctamente el intercambio de valores entre *a* y *b*, usando una tercera variable *c*, en la forma en que normalmente se hace en todo lenguaje:

```
# valores iniciales…
a = 12
b = 4

# secuencia de intercambio al estilo clásico…
c = a
a = b
b = c
```

```
# visualización de valores intercambiados…
print('Valor final de a: ', a)
print('Valor final de b: ', b)
```

Aun cuando lo anterior funciona, el hecho es que en Python se puede hacer el intercambio de valores recurriendo a la asignación de *tuplas* en forma directa. La siguiente instrucción en Python, hace el intercambio de valores entre las variables *a* y *b* sin tener que recurrir en forma explícita a una variable auxiliar (ya que en realidad, *la variable auxiliar es la tupla que Python crea al evaluar el miembro derecho*) (Asegúrese de comprender la secuencia que sigue, haciendo una gráfica del estado de las variables similar al de la *Figura 4* en cada momento):

```
# valores iniciales…
a = 12
b = 4

# intercambio al estilo Python…
a, b = b, a

# visualización de valores intercambiados…
print('Valor final de a: ', a)
print('Valor final de b: ', b)
```

2.] Cadenas de caracteres: Elementos adicionales.

Las *cadenas de caracteres* o *strings* constituyen un tipo de dato fundamental en todo lenguaje de programación. Como sabemos, en Python un *string* se define como una *secuencia inmutable de caracteres* y por lo tanto, le son aplicables todos los operadores, funciones y métodos propios del manejo de secuencias [1].

Un *literal* (o sea, una *constante*) de tipo string es una secuencia de caracteres delimitada por comillas dobles o por comillas simples (o *apóstrofos*). Hemos visto que se pueden usar indistintamente las comillas dobles o las comillas simples, pero si se comenzó con un tipo, se debe cerrar con el mismo tipo de comilla:

```
nombre = "Pedro"
provincia = 'Córdoba'
```

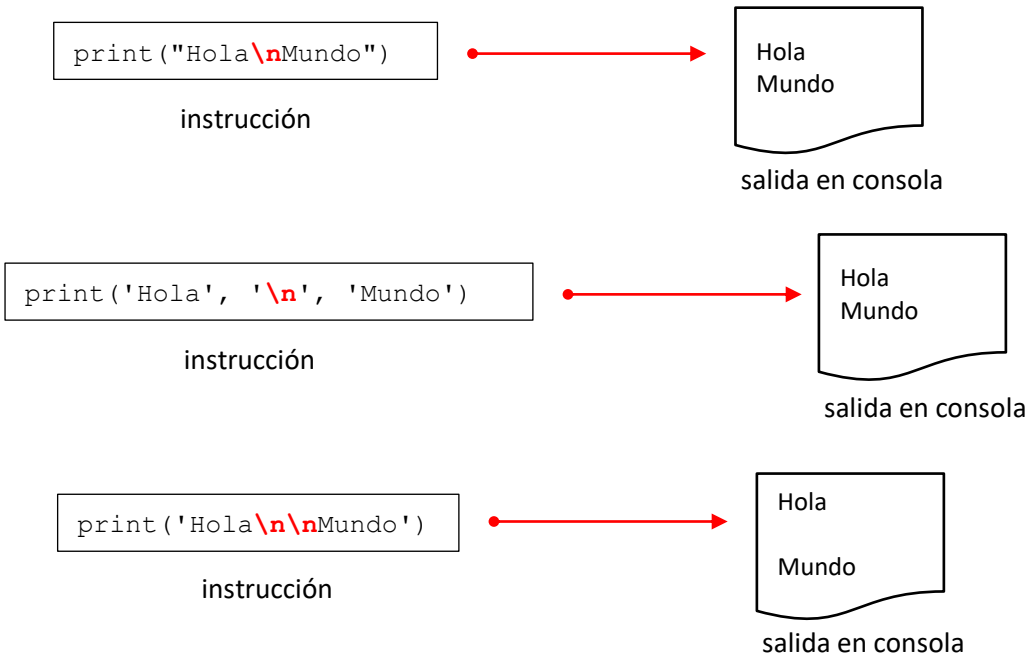
Lo anterior es útil, por ejemplo, si se quiere almacenar *literalmente* una comilla doble o simple como parte de una cadena constante: si el delimitador usado fue comilla doble, entonces la comilla simple puede usarse dentro de la cadena como caracter directo, y viceversa:

```
apellido = "O'Donnell"
autor = 'Anthony "Tony" Hoare'
```

Tanto en Python como en otros lenguajes (Java, C, C++, por ejemplo) se pueden usar caracteres especiales llamados *caracteres de escape* o *caracteres de control*. Un caracter de escape se forma con un símbolo de barra (\) seguido de algún caracter especial que actúa en forma de código: cuando Python encuentra una secuencia que corresponde a un caracter de escape dentro de un literal de tipo string, interpreta a la secuencia *caracter* como un solo y único caracter.

En algunos casos, el caracter de escape provoca algún efecto en la salida por consola (y no exactamente la visualización del caracter). Por ejemplo, el caracter "\n" produce el efecto de un salto de línea en la salida por consola cuando aparece dentro de una cadena (vea los ejemplos en la gráfica siguiente):

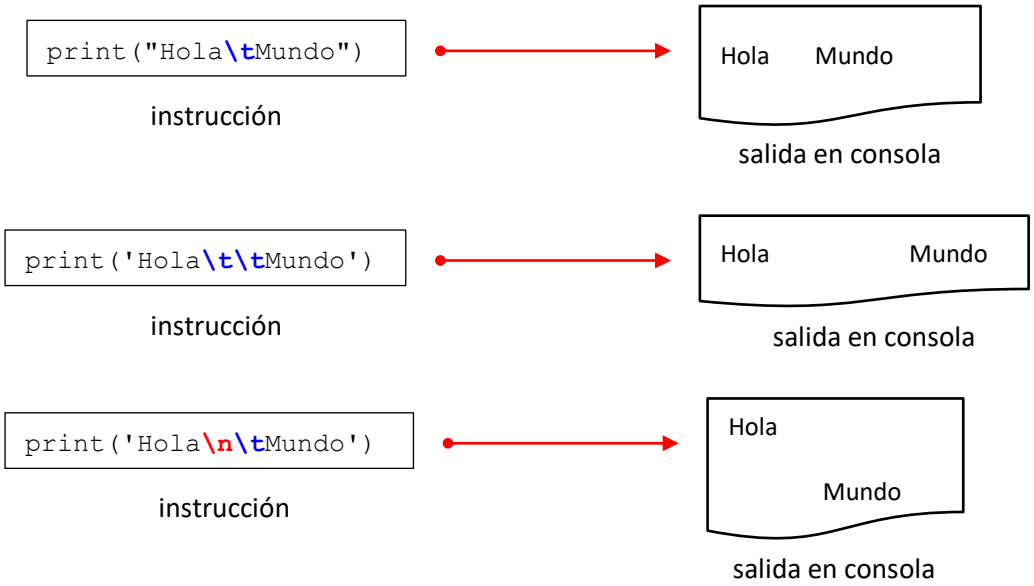
Figura 5: Efectos de uso del caracter de salto de línea.



Como se puede ver en el tercer ejemplo de la *Figura 5*, los caracteres de control pueden combinarse: dos \n seguidos, producen dos saltos de línea en la consola de salida.

Por otra parte, el caracter de control \t provoca un espaciado en la consola de salida, equivalente a una "sangría" (o "espaciado horizontal" o "Tab") (normalmente, 4 espacios en la misma línea, a partir del cursor). Y todos pueden combinarse, repitiendo varios del mismo tipo o agrupando dos o más de tipos diferentes (ver *Figura 6*):

Figura 6: Efectos de uso del caracter de espaciado horizontal.



Los caracteres de escape \' y \" representan *literalmente* comillas simples o comillas dobles y permiten entonces su uso dentro de cadenas en forma directa, sin importar si estas cadenas fueron abiertas con el mismo tipo de delimitador [1]:

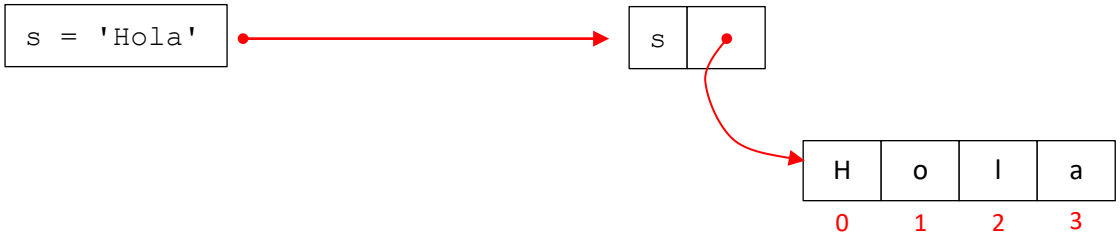
```
apellido = 'O\'Donnell'  # almacena:  O'Donell
```

```
autor = "Anthony \"Tony\" Hoare"  # almacena: Anthony "Tony" Hoare
```

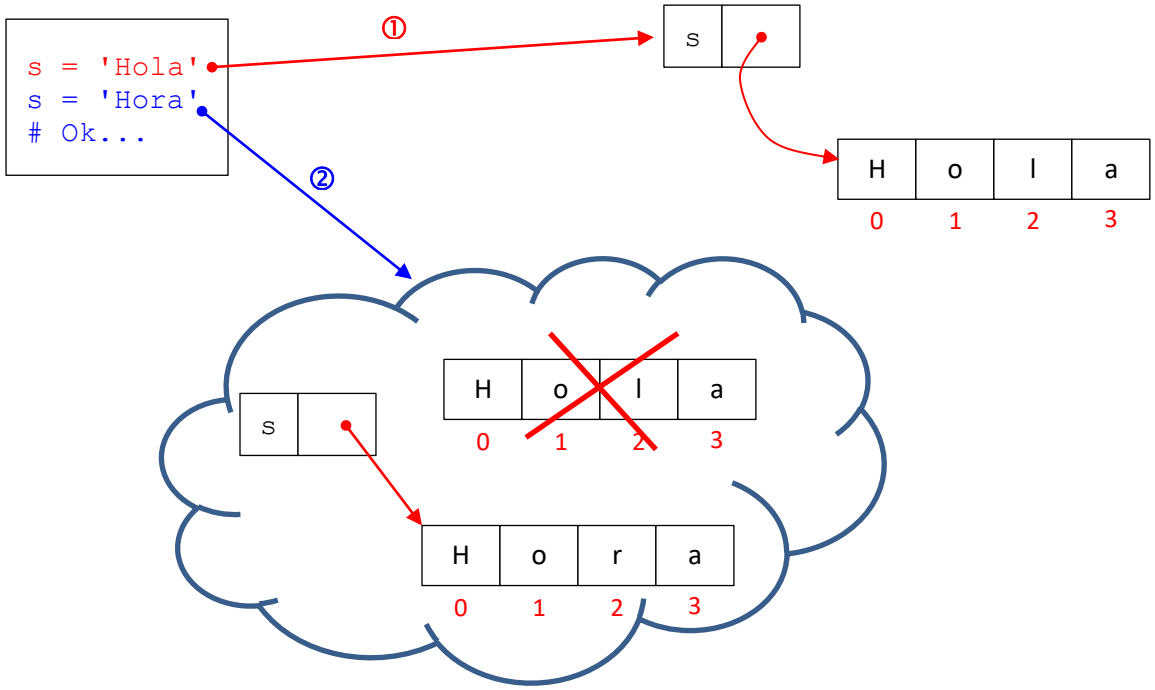
También sabemos que los caracteres individuales de una cadena se almacenan como una secuencia con índices numerados desde cero en adelante, y que *el índice puede usarse para acceder a esos caracteres, sin modificarlos* (recuerde que una cadena es una secuencia ***inmutable***):

```
s = 'Hola'
print('Primeras dos letras: ', s[0], s[1])
s[2] = 'r'  # error...
```

Una variable de tipo cadena (y cualquier variable en general) contiene en realidad la *dirección del byte de memoria donde la cadena comienza* (y se dice entonces que la variable *apunta* a esa dirección o a esa cadena):



Debe quedar claro que lo que **no** puede hacerse es *cambiar el valor de un casillero de una cadena ya creada y almacenada en memoria*. En el ejemplo anterior, el intento de cambiar la "l" en la casilla s[2] por una "r" (o sea, s[2] = "r") provoca un error de intérprete. *Pero no hay ningún inconveniente en que una variable que apuntaba a una cadena, pase a apuntar a otra diferente* (en este caso, se crea la nueva cadena y la anterior se pierde):



Para finalizar estar sección, resaltamos un hecho interesante: los operadores *suma (+)* y *producto (*)* en Python también pueden usarse con cadenas de caracteres. El primero permite lo que se conoce como la *concatenación* o *unión* de dos o más cadenas: se crea una *nueva*

cadena que contiene a todas las cadenas originales, en el mismo orden en que se unieron y una detrás de la otra [1]:

```
nombre = 'Guido'
apellido = "van Rossum"
completo = nombre + " " + apellido
print("Nombre completo: ", completo)
# muestra: Guido van Rossum
```

Y el segundo permite *repetir* una cadena varias veces concatenando el resultado:

```
replicado = nombre * 4
print("Nombre repetido: ", replicado)
# muestra: GuidoGuidoGuidoGuido
```

3.] Funciones comunes de la librería estándar de Python.

Los operadores aritméticos de Python que hasta aquí hemos visto permiten el planteo directo de muchos cálculos y fórmulas comunes y generales. No obstante, y en forma esperable, Python provee un amplio conjunto de *funciones internas*, listas para usar, que están siempre disponibles en un script o programa sin necesidad de declaraciones ni instrucciones adicionales. La lista completa de estas funciones puede verse en la documentación de ayuda de Python [3] que se instala junto con el lenguaje, pero aquí extraemos algunas de las más comunes, para facilitar el avance:

Figura 7: Tabla de las principales funciones de la librería estándar de Python.

Función	Descripción	Ejemplo de uso
abs(x)	Retorna el valor absoluto del parámetro **x**.	x = -23 y = abs(x) print(y) # muestra: 23
bin(x)	Obtiene la conversión a binario (base 2) del valor **x**, en formato de cadena cuyos dos primeros caracteres son '0b', indicando que lo que sigue es una secuencia en binario.	x = 8 s = bin(x) print(s) # muestra: 0b1000
chr(i)	Retorna el caracter (en formato de string) que corresponde al valor Unicode número **i**.	i = 65 c = chr(i) print(c) # muestra: A
divmod(a, b)	Retorna el cociente y el resto de la división entera entre los parámetros **a** y **b**.	a, b = 14, 4 c, r = divmod(a,b) print(c, r) # muestra: 3 2
float(x)	Convierte la cadena **s** a un número en coma flotante. Hemos usado esta función combinada con la función *input()* para cargar números flotantes desde teclado.	s = '23.45' y = float(s) print(y) # muestra: 23.45
hex(x)	Obtiene la conversión a hexadecimal (base 16) del valor **x**, en formato de cadena cuyos dos primeros caracteres son '0x', indicando que lo que sigue es una secuencia en hexadecimal.	x = 124 s = hex(x) print(s) # muestra: 0x7c
input(p)	Obtiene una cadena de caracteres desde la entrada estándar y la retorna. La cadena **p** es visualizada a modo de "prompt" mientras se espera la carga.	Ver los ejemplos de uso de carga por teclado en las fichas 1 y 2 del curso.

int(x)	Convierte la cadena **s** a un número entero. Hemos usado esta función combinada con la función *input()* para cargar números enteros desde teclado.	s = '1362' y = int(s) print(y) # muestra: 1362
len(s)	Retorna la longitud del objeto **t** tomado como parámetro (un string, una lista, una tupla o un diccionario). La longitud es la *cantidad de elementos* que tiene el objeto **t**.	t = 12, 45, 73 n = len(t) print(n) # muestra: 3
max(a, b, *args)	Retorna el mayor de los parámetros tomados. Notar que admite una *cantidad arbitraria* de parámetros: **max(a,b)** retorna el mayor entre **a** y **b**, pero **max(a,b,c,d)** retorna el mayor entre **a**, **b**, **c** y **d**.	a, b = 6, 3 my = max(a, b) print(my) # muestra: 6
min(a, b, *args)	Retorna el menor de los parámetros tomados. Notar que admite una *cantidad arbitraria* de parámetros: **min(a,b)** retorna el menor entre **a** y **b**, pero **min(a,b,c,d)** retorna el menor entre **a**, **b**, **c** y **d**.	a, b = 6, 3 mn = min(a, b) print(mn) # muestra: 3
oct(x)	Obtiene la conversión a octal (base 8) del valor **x**, en formato de cadena cuyos dos primeros caracteres son '0o', indicando que lo que sigue es una secuencia en octal.	x = 124 s = oct(x) print(s) # muestra: 0o174
ord(c)	Retorna el entero Unicode que representa al caracter **c** tomado como parámetro.	c = 'A' i = ord(c) print(i) # muestra: 65
pow(x, y)	Retorna el valor de **x** elevado a la **y**. Note que lo mismo puede hacerse con el operador "doble asterisco" (**). En Python: **r = pow(x, y)** es lo mismo que **r = x**y**.	x, y = 2, 3 r = pow(x, y) print(r) # muestra: 8
print(p)	Muestra el valor **p** en la consola estándar.	Ver los ejemplos de uso de salida por consola estándar en las fichas 1 y 2 del curso.
round(x, n)	Retorna el número flotante **x**, pero redondeado a **n** dígitos a la derecha del punto decimal. Si **n** se omite retorna la parte entera de **x** (como int). Si **n** es cero, retorna la parte entera de **x** (pero como float): **round(3.1415)** retornará 3 (int), pero **round(3.1415, 0)** retornará 3.0 (float).	x = 4.1485 r = round(x, 2) print(r) # muestra: 4.15
str(x)	Retorna una versión en forma de cadena de caracteres del objeto **x** (es decir, retorna la *conversión a string* de **x**).	x = 2, 4, 1 s = str(x) print(s) *# muestra: (2, 4, 1)*

4.] Aplicaciones y casos de análisis.

Toda esta sección está dedicada al estudio y resolución de problemas simples basados sólo en estructuras secuenciales de instrucciones. Cada problema será planteado en la forma que hemos sugerido hasta ahora: comprensión general del enunciado, planteo de un algoritmo mediante un diagrama de flujo y mediante un pseudocódigo, y finalmente el desarrollo del programa en Python. La numeración de cada problema, sigue en forma correlativa a los ya planteados en la Ficha 2 y en lo que va de esta Ficha 3.

Problema 5.) *La fuerza de atracción entre dos masas **m1** y **m2** separadas por una distancia **d**, está dada por la siguiente fórmula[2] de la mecánica clásica:*

$$F = G * \left(\frac{m1 * m2}{d^2}\right)$$

*con G = 6.673 * 10^{-8} (constante de gravitación universal)*

Escribir un programa que cargue las masas m1 y m2 de dos cuerpos y la distancia d entre ellos y obtenga y muestre el valor de la intensidad de la fuerza de atracción entre esos cuerpos.

a.) Identificación de componentes:

- **Resultados:** Intensidad de la fuerza de atracción entre dos cuerpos[3]. (*f*: número real)
- **Datos:** Masa de los dos cuerpos. (*m1, m2*: números reales)
  Distancia entre ellos. (*d*: número real)
- **Procesos:** Problema de *física*. El enunciado mismo del problema ya brinda la fórmula necesaria para el cálculo, la que sólo debe plantearse adecuadamente para ser ejecutada en un programa. El valor de la constante *G* viene dado, y por lo tanto no debe cargarse por teclado: simplemente se asigna el valor en una variable auxiliar *G* y se usa sin más. Esa asignación y la fórmula replanteada (con sintaxis de Python) sería:

```
G = 6.673 * pow(10, -8)
F = (G * m1 * m2) / d ** 2
```

Por razones de sencillez, se supone aquí que los datos serán ingresados de forma que las unidades de cada magnitud sean las correctas (por ejemplo, las masas en kilogramos y la distancia en metros) y por lo tanto el programador no deberá incluir fórmulas de conversión. También suponemos que las masas y la distancia serán cargadas como positivas, pero como siempre hemos indicado, en una situación real el programador debería incluir algún tipo de validación al cargar esos datos y para evitar que se ingresen negativos o cero.

b.) Planteo del algoritmo: Mostramos tanto el *diagrama de flujo* como el *pseudocódigo* en la Figura 8 de la página siguiente.

c.) Desarrollo del programa: En base al diagrama y/o el pseudoccódigo, el script o programa se deduce en forma simple:

```
__author__ = 'Cátedra de AED'


# asignación de constantes...
G = 6.673 * pow(10,-8)

# título general y carga de datos...
```

[2] Este problema fue sugerido por la *Ing. Marina Cardenas*, docente de esta Cátedra, quien también desarrolló el script en Python para resolverlo.

[3] Respecto de la gravedad, la forma en que decrece a medida que nos alejamos de la Tierra, y los problemas que causa su ausencia en un viaje al espacio, podemos sugerir la muy taquillera película *Gravity* (*Gravedad*) del año 2013, dirigida por *Alfonso Cuarón* y protagonizada por *Sandra Bullock* y *George Clooney*. Un poco más vieja, pero muy vigente, es la famosísima *Apollo 13* de 1995, dirigida por *Ron Howard* y protagonizada por *Tom Hanks*.

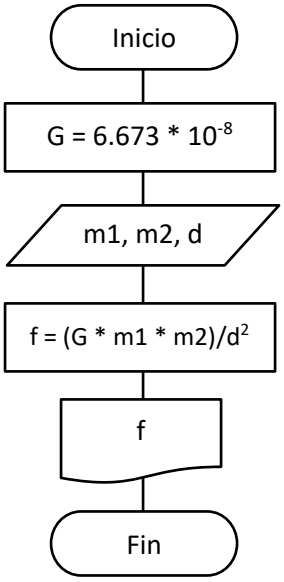
```
print('Ejemplo 4 - Cálculo de la fuerza de atracción')
m1 = float(input('Masa del primer cuerpo: '))
m2 = float(input('Masa del segundo cuerpo: '))
d = float(input('Distancia entre ambos: '))

# procesos...
f = (G * m1 * m2) / d**2

# visualización de resultados...
print('Fuerza de atracción:', f)
```

En este programa aparecen cinco variables, y en cuatro de ellas (*m1*, *m2*, *d* y *f*) hemos usado nombres en minúscula, mientras que la restante (*G*) tiene su nombre en mayúscula. De acuerdo al lenguaje que se use, existen muchas convenciones en cuanto al estilo de designación de nombres de variables. En Python no hay una recomendación especial, y sólo se pide que sea cual sea la que siga el programador sea claramente distinguible. En algunos lenguajes se recomienda que todo nombre de variable comience con minúscula. En otros, que comience con mayúscula. En otros, que se usen siempre minúsculas… Y así por el estilo.

Figura 8: Diagrama de flujo y pseudocódigo del problema de cálculo de la fuerza de atracción.




```
Algoritmo:
    1. Sea G = 6.673 * 10-8
    2. Cargar m1: masa del primer cuerpo
    3. Cargar m2: masa del segundo cuerpo
    4. Cargar d: distancia entre ellos
    5. Sea f = (G * m1 * m2) / d2
    4. Mostrar f: intensidad de la fuerza
```

En la última sección de esta Ficha 3 ("Convenciones de estilo…", página 67) con la que se cierra la Unidad 1 del programa de AED, hemos incluido una descripción más profunda del tema de los estilos de desarrollo de código fuente. Por ahora, y sólo para cerrar el análisis del programa del cálculo de la intensidad de la fuerza de atracción, digamos que en el desarrollo de las Fichas de Estudio de este curso, y en los ejemplos de ejercicios y programas resueltos que se entreguen a los estudiantes, aplicaremos las siguientes convenciones en cuanto a nombres de variables[4]:

- El nombre una variable se escribirá siempre en minúsculas (Ejemplos: *sueldo*, *nombre*, *lado*).
- Cuando se quiera que el nombre de una variable agrupe dos o más palabras, usaremos el *guión bajo* como separador (Ejemplos: *mayor_edad*, *horas_extra*).
- Excepcionalmente designaremos en *mayúsculas sostenidas* a alguna variable que represente algún valor muy conocido y normalmente escrito en mayúsculas (Ejemplos: *G* (por la constante

[4] No obstante, los estudiantes (e incluso sus profesores) podrán usar otras convenciones reconocidas si las prefieren. Sólo se sugiere que se mantengan consistentes con ellas.

de gravitación universal que vimos en el programa anterior), o *PI* (por el número *pi*), o *IVA* (por nuestro bendito impuesto al valor agregado)).

Veamos otro ejemplo de problema simple:

Problema 6.) *La dirección de la carrera de Ingeniería en Sistemas de la UTN Córdoba necesita un programa que permita cargar el nombre de un estudiante de quinto año, el nombre del profesor responsable de la Práctica Profesional Supervisada de ese estudiante, y el promedio general (con decimales) del estudiante en su carrera. Una vez cargados los datos, se pide simplemente mostrarlos en la consola de salida a modo de verificación, pero de forma que el nombre del practicante vaya precedido de la cadena "est." y el nombre del profesor supervisor se preceda con "prof.". El promedio del alumno debe mostrarse redondeado, sin decimales, en formato entero.*

a.) Identificación de componentes:

- **Resultados:** "est. " + Nombre del practicante. (*nom_est_final*: string)
  "prof. " + Nombre del supervisor. (*nom_pro_final*: string)
  Promedio entero redondeado del practicante. (*prom_final*: int)
- **Datos:** Nombre del practicante. (*nom_est_orig*: string)
  Nombre del supervisor. (*nom_pro_orig*: string)
  Promedio real del practicante. (*prom_orig*: número real)
- **Procesos:** Problema de *procesamiento de cadenas y redondeo de números, en un contexto de gestión académica*. Esencialmente, se trata de mostrar los mismos datos que se cargaron, pero modificándolos ligeramente para obtener el formato de salida pedido.

  En principio, agregar el prefijo "est. " al nombre del estudiante es una simple operación de *concatenación* o *unión* de cadenas, cuyo resultado podría volver a almacenarse en la misma variable donde viene el nombre original del estudiante:

  nom_est_orig = 'est. ' + nom_est_orig

  Sin embargo, esto haría que el dato original se pierda. En este problema en particular no parece un inconveniente mayor ya que el enunciado no pide volver a usar ese dato en un proceso posterior. Pero en general será recomendable que el programador preserve los datos originales, de forma de poder volver a usarlos más adelante si los necesitase. Por lo tanto, se deben almacenar los resultados de las concatenaciones en dos variables nuevas (una para el practicante y otra para el profesor supervisor):

  nom_est_final = 'est. ' + nom_est_orig
  nom_pro_final = 'prof. ' + nom_pro_orig

  Y en cuanto al promedio, el dato viene como número real y se está pidiendo mostrarlo como entero redondeado, lo cual puede hacerse en forma directa con la función *round()* provista por Python y que presentamos en la Ficha 2:

  prom_final = round(prom_orig)

b.) Planteo del algoritmo: Mostramos tanto el *diagrama de flujo* como el *pseudocódigo* a en la Figura 9 de la página siguiente.

c.) Desarrollo del programa: En base al diagrama y/o el pseudocódigo, el script o programa se deduce en forma simple:

```
__author__ = 'Cátedra de AED'


# título general y carga de datos...
```

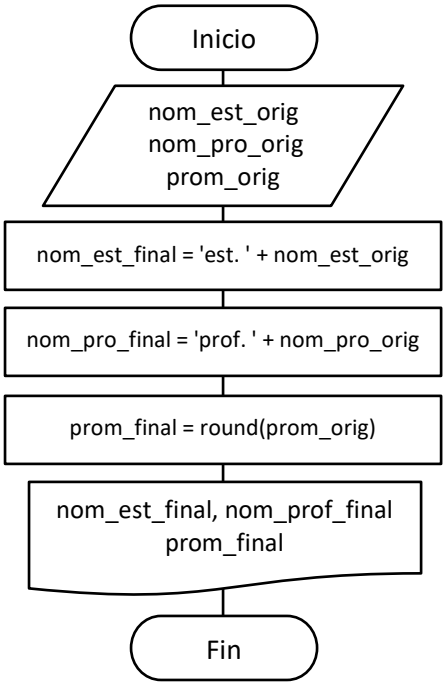
```
print('Ejemplo 6 - Datos de Práctica Profesional Supervisada')
nom_est_orig = input('Nombre del estudiante: ')
prom_orig = float(input('Promedio general: '))
nom_pro_orig = input('Nombre del profesor: ')

# procesos...
nom_est_final = 'est. ' + nom_est_orig
nom_pro_final = 'prof. ' + nom_pro_orig
prom_final = round(prom_orig)

# visualización de resultados...
print('Practicante:', nom_est_final, '\t-\tPromedio:', prom_final)
print('Supervisor:', nom_pro_final)
```

Todas las variables de este script han sido designadas con nombres que siguen el estilo indicado al final del problema anterior: todo el nombre en letras minúsculas y el uso del guión bajo como separador.

Figura 9: Diagrama de flujo y pseudocódigo del problema de visualización de datos de un alumno.




```
Algoritmo:
   1. Cargar nom_est_orig: nombre del estudiante
   2. Cargar nom_prof_orig: nombre del profesor
   3. Cargar prom_orig: promedio del estudiante
   4. Sea nom_est_final = 'est. ' + nom_est_orig
   5. Sea nom_prof_final = 'prof. ' + nom_prof_orig
   6. Sea prom_final = round(prom_orig)
   7. Mostrar nom_est_final
   8. Mostrar nom_prof_final
   9. Mostrar prom_final
```

El siguiente problema se sugiere para aplicar propiedades del resto de la división:

Problema 7.) *Una pequeña oficina de correos dispone de cinco casillas o boxes para guardar las cartas que debe despachar. Cada casilla (que puede contener muchas cartas) está numerada con números correlativos del 0 al 4. Cada carta que se recibe tiene un código postal numérico, y en base a ese código el despachante debe determinar en qué box guardará la carta, pero de tal forma que el mismo sistema sirva luego para saber en qué caja buscar una carta, dado su código postal. Diseñe un esquema de distribución que permita cumplir este requerimiento, cargando por teclado el código postal de tres cartas y mostrando en qué casilla será almacenada cada una.*

a.) Identificación de componentes:

- **Resultados:** Números de las casillas donde serán almacenadas las tres cartas recibidas. (*n1, n2, n3*: números enteros)
- **Datos:** Códigos postales de tres cartas (*c1, c2, c3*: números enteros)

- **Procesos:** Problema de *procesamiento y aplicación de relaciones de congruencia, en un contexto de correo postal*. Está claro que el enunciado propone un nuevo desafío: diseñar un esquema de distribución de cartas, pero sin dar muchas pistas respecto de cómo hacerlo. No hay fórmulas, ni pasos evidentes indicados en ese enunciado. Sólo hay un requerimiento. Y este tipo de situaciones son las que más típicamente enfrentan los programadores en su trabajo: entienda que las personas que soliciten sus servicios no saben nada de programación ni de algoritmos… sólo pueden decirle qué datos le darán y qué tipo de resultados esperan obtener… y el resto es tarea del profesional de la programación.

  Veamos: El problema es tratar de asignar a cada código postal posible un número de casillero, sabiendo que estos últimos están todos en el rango del intervalo entero [0, 4]. Una primera idea (errónea…) podría ser tomar el último dígito del código postal y usar ese dígito como número de casillero. Pero puede verse fácilmente que esta estrategia no funciona: si el código postal termina con cualquier dígito mayor a 4 (como 5347, o 2238), no habrá un casillero identificado con ese dígito y el sistema de distribución fallará[5].

  Una idea mejor surge de observar que el rango de posibles números de casilleros *es exactamente igual al conjunto de posibles restos de la división por 5*. Entonces, si para cada código postal simplemente se toma el resto de dividir por 5, se obtendrá inexorablemente un número entre 0 y 4 que puede ser asignado como número de casillero para la carta en cuestión. Así, la carta con código postal 5347 será asignada al casillero 2 (ya que 5347 % 5 = 2) y también irá al casillero 2 una carta cuyo código postal sea, por ejemplo, 3672 (ya que 3672 % 5 = 2).

  En definitiva, y matemáticamente hablando [3], cada casillero puede ser equiparado a $\mathbf{Z5_n}$: la *clase de congruencia (módulo 5) con el subíndice n igual al resto de dividir por 5*, como vimos en la Ficha 2. Los números 5347 y 3672 pertenecen a $\mathbf{Z5_2}$ y por ello las cartas con esos códigos postales van al box 2. Los números 8761 y 9236 pertenecen a $\mathbf{Z5_1}$ y por eso las cartas con esos códigos irán al box 1.

  Conclusión: nuestro algoritmo sólo debe tomar el resto de dividir por *n* = 5 a cada código postal y mostrar ese resto. El número 5 (la cantidad de casilleros) puede asignarse previamente como una constante inicial *n*, lo cual le da al código fuente algo de flexibilidad si luego el programador debe modificarlo: si el número de casillas pasa de 5 a 13, por ejemplo, el programador sólo deberá cambiar el valor de *n* y el resto del programa seguirá funcionando correctamente.

b.) Planteo del algoritmo: Mostramos tanto el *diagrama de flujo* como el *pseudocódigo* en la Figura 10 de la página siguiente.

c.) Desarrollo del programa: En base al diagrama y/o el pseudoccódigo, el script o programa se deduce en forma simple:

```
__author__ = 'Cátedra de AED'


# asignación de constantes...
n = 5

# título general y carga de datos...
print('Ejemplo 7 - Distribución de cartas')
c1 = int(input('Primer código postal: '))
```

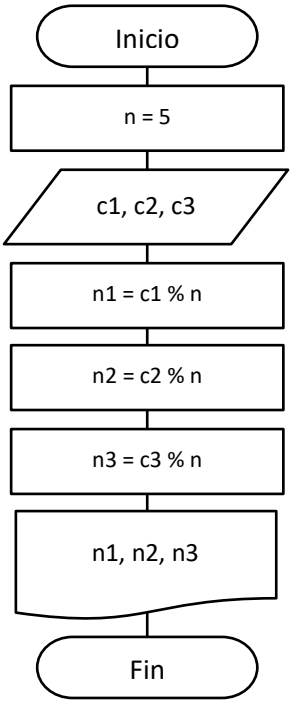
[5] Aceptamos que *sólo funcionará* si el programador tuviese la suerte de contar con exactamente 10 casilleros en lugar de 4.

```
c2 = int(input('Segundo código postal: '))
c3 = int(input('Tercer código postal: '))

# procesos...
n1 = c1 % n
n2 = c2 % n
n3 = c3 % n

# visualización de resultados...
print('Código postal:', c1, '\tasignado al box: ', n1)
print('Código postal:', c2, '\tasignado al box: ', n2)
print('Código postal:', c3, '\tasignado al box: ', n3)
```

Figura 10: Diagrama de flujo y pseudocódigo del problema de distribución de cartas.




```
Algoritmo:
   2. Sea n = 5
   1. Cargar c1, c2, c3: códigos postales de 3 cartas
   2. Sea n1 = c1 % n
   3. Sea n2 = c2 % n
   4. Sea n3 = c3 % n
   5. Mostrar n1, n2, n3: números de casillas asignadas
```

La idea general usada en este problema para calcular el número del box que corresponde a cada carta es la base de una técnica de organización de datos para búsqueda rápida *muy eficiente*, conocida como *búsqueda por dispersión de claves* (o como se la designa en inglés: *hashing*[6]). El estudio de esta técnica está fuera del alcance de este curso: se lleva a cabo en asignaturas o cursos avanzados de estructuras de datos, más adelante en la Carrera.

5.] Convenciones de estilo de escritura de código fuente en Python: la guía PEP.

Llegados a este punto parece prudente introducir algunos conceptos referidos a *convenciones y reglas de estilo* cuando se programa en Python. Entre la muy extensa documentación de ayuda del lenguaje, se incluyen algunos documentos oficiales que contienen sugerencias para que los programadores escriban código fuente consistente, entendible y claro para otros programadores. Entre esos documentos se destaca claramente el conocido como *PEP*

[6] En este contexto, la palabra *hashing* se puede entender como "*desmenuzamiento*" o "*hacer picadillo*" o "*convertir en migajas*". Justamente, la acción de tomar un número o clave original y obtener a partir de él un resto simple, da la idea de haber *desmigajado* ese número para tomar sólo una pequeña parte del mismo (el resto) para continuar el proceso.

(iniciales de *Python Enhancement Proposal* o *Propuesta de Mejora de Python*)[7], que contiene muchos capítulos dependiendo de cuál sea el tema o aspecto del lenguaje sobre el que se están fijando convenciones [4].

En particular, el documento o capítulo *PEP 8 – Style Guide for Python Code*[8] (o *Guía de Estilos para Código Python*) se centra específicamente en recomendaciones de estilo de escritura de código fuente en Python, y desde *PEP 8* tomamos las recomendaciones que siguen [5]. Por supuesto, la guía *PEP 8* es mucho más extensa, y lo que mostramos en este resumen es una selección de las principales convenciones de estilo aplicables a un curso que lleva sólo tres semanas de desarrollo. Por supuesto, recomendamos encarecidamente al estudiante que se acostumbre a consultar las guías *PEP* por su propia cuenta, para mantenerse actualizado en cuanto a recomendaciones, reglas y convenciones.

Todo lenguaje tiene sus convenciones y recomendaciones de estilo. Entre los programadores Python se habla de *Filosofía Python* para referirse a la forma de hacer las cosas en Python. En ese contexto, si un código fuente respeta las convenciones de legibilidad y sencillez de la *Filosofía Python*, se dice que es un *código pythónico* (por *pythonic code*), mientras que lo opuesto (complejo y poco claro) sería un *código no pythónico* (por *unpythonic*). Por lo pronto, entonces, comenzamos transcribiendo una famosa tabla informal de principios pythónicos que fueron presentados graciosamente en forma de aforismos por el desarrollador *Tim Peters* en el documento *PEP 20* (conocido como *El Zen de Python*[9]*)* [6]:

Figura 11: El Zen de Python…

- ✓ Bello es mejor que feo.
- ✓ Explícito es mejor que implícito.
- ✓ Simple es mejor que complejo.
- ✓ Complejo es mejor que complicado.
- ✓ Plano es mejor que anidado.
- ✓ Disperso es mejor que denso.
- ✓ La legibilidad cuenta.
- ✓ Los casos especiales no son tan especiales como para quebrantar las reglas.
- ✓ Aunque lo práctico le gana a la pureza.
- ✓ Los errores nunca deberían dejarse pasar silenciosamente.
- ✓ A menos que hayan sido silenciados explícitamente.
- ✓ Frente a la ambigüedad, rechaza la tentación de adivinar.
- ✓ Debería haber una -y preferiblemente sólo una- manera obvia de hacerlo.
- ✓ Aunque esa manera puede no ser obvia al principio a menos que usted sea holandés.
- ✓ Ahora es mejor que nunca.
- ✓ Aunque *nunca* es a menudo mejor que *ya mismo*.
- ✓ Si la implementación es difícil de explicar, es una mala idea.
- ✓ Si la implementación es fácil de explicar, puede que sea una buena idea.
- ✓ Los espacios de nombres (*namespaces*) son una gran idea ¡Hagamos más de esas cosas!

[7] Disponible en forma completa en https://www.python.org/dev/peps/.

[8] Disponible en https://www.python.org/dev/peps/pep-0008/.

[9] Por extraño que parezca, *PEP 20* sólo contiene la tabla de aforismos que hemos transcripto (aunque, por supuesto, en inglés). Disponible en https://www.python.org/dev/peps/pep-0020/. Y aún más curioso: estos aforismos están incluidos en el SDK de Python a modo de *Easter Egg* (*Huevo de Pascua*): un tipo de mensaje oculto en un programa que sólo aparece si se sabe cómo encontrarlo (normalmente con alguna combinación no documentada de teclas, o con alguna instrucción especial en un programa). En el caso del *Zen*, lo verá si abre el shell de Python y simplemente ejecuta la instrucción >>> *import this* (no incluya los signos >>> que conforman el prompt del shell).

Finalmente, presentamos aquí (de la guía *PEP 8*) un subconjunto de las convenciones de estilo de escritura de código en Python más aplicables en este momento del curso (insistimos: no dude en consultar la guía *PEP 8* completa para ampliar estas referencias) [5]:

- Para indentar use 4(cuatro) espacios en lugar de *tabs* (tabuladores). El uso de tabuladores introduce confusión, y con 4 espacios de indentación tendrá el equilibrio justo entre "poco" y "demasiado".
- Corte sus líneas de código en no más de 79 caracteres por línea. Esto le ayudará a mantener visible gran parte del código fuente en pantallas pequeñas, y le permitirá mostrar varios archivos de código fuente uno al lado del otro en pantallas grandes.
- Use líneas en blanco para separar funciones, clases y largos bloques de código dentro de una función o una secuencia (veremos la forma de desarrollar funciones más adelante en el curso).
- Cuando sea posible, coloque sus comentarios en líneas específicas para esos comentarios (de ser posible, no los agregue en la misma línea de una instrucción).
- Utilice cadenas de documentación (*dostrings*) [2]: mantenga actualizada la documentación de sus programas (veremos esto más adelante el curso).
- Coloque espacios alrededor de los operadores en un expresión, y después de las comas en una enumeración, pero no inmediatamente a los lados de cada paréntesis cuando los use: `a = f(1, 2) + g(3, 4).`
- No utilice codificaciones de caracteres demasiado extrañas si está pensando en que su código fuente se use en contextos internacionales. En cualquier caso, el default de Python (que es UTF-8) o incluso el llamado *texto plano* (ASCII) funciona mejor.
- Del mismo modo, evite el uso de caracteres no-ASCII en el nombre de un identificador (por ejemplo, en el nombre de una variable) si existe aunque sea una pequeña posibilidad de que su código fuente sea leído y mantenido por personas que hablen en otro idioma.
- En cuanto al uso de comillas dobles o simples para delimitar cadenas de caracteres, no hay una recomendación especial. Simplemente, mantenga la consistencia: si comenzó con una forma de hacerlo, apéguese a ella y no la cambie a cada momento. Cuando necesite incluir un tipo de comilla (simple o doble) en una cadena, use la otra para delimitarla (en lugar de caracteres de control que restan legibilidad).
- Si tiene operadores con diferentes prioridades, considere agregar espacios alrededor de los que tengan menor prioridad, y eliminar los espacios en los que tengan prioridad mayor. Use el sentido común. Sin embargo, nunca coloque más de un espacio y siempre coloque la misma cantidad de espacios antes y después del mismo operador.

```
Sí:
    m = p + 1
    t = x*2 - 1
    h = x*x + y*y
    c = (a+b) * (a-b)
```

```
No:
    p=p+1
    t= x*2-1
    h = x * x + y * y
    c = (a + b) * (a - b)
```

- Como ya se indicó, en Python no hay una recomendación especial en cuanto a convenciones de nombres o identificadores. En general, use sentido común: sea cual sea la convención que siga el programador, debe ser claramente distinguible y mantenerse en forma coherente. Otra vez, digamos que en el desarrollo de las Fichas de Estudio de

este curso y en los ejemplos de ejercicios y programas resueltos que se entreguen a los estudiantes, aplicaremos las siguientes convenciones en cuanto a nombres de variables (y de nuevo, aclaramos que los estudiantes e incluso sus profesores podrán usar otras convenciones reconocidas si las prefieren. Sólo se sugiere que se mantengan consistentes con ellas):

- El nombre una variable se escribirá siempre en minúsculas (Ejemplos: *sueldo*, *nombre*, *lado*).
- Cuando se quiera que el nombre de una variable agrupe dos o más palabras, usaremos el *guión bajo* como separador (Ejemplos: *mayor_edad*, *horas_extra*).
- Excepcionalmente designaremos en *mayúsculas sostenidas* a alguna variable que represente algún valor muy conocido y normalmente escrito en mayúsculas (Ejemplos: *PI* (por el número *pi*), o IVA (por nuestro bendito impuesto al valor agregado)).

- Nunca utilice la l ('letra ele' minúscula) ni la O ('letra o' mayúscula) ni la I ('letra i' mayúscula) como nombre simple de una variable: en algunas fuentes de letras esos caracteres son indistinguibles de los números 1 y 0 y obviamente causaría confusión.

Bibliografía

[1] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: https://docs.python.org/3/.

[2] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.

[3] E. Gentile, Notas de Álgebra I, Buenos Aires: Eudeba, 1976.

[4] Python Software Foundation, "PEP 0 - Index of Python Enhancement Proposals (PEPs)," 2015. [Online]. Available: https://www.python.org/dev/peps/.

[5] Python Software Foundation, "PEP 8 - Style Guide for Python Code," 2015. [Online]. Available: https://www.python.org/dev/peps/pep-0008/.

[6] Python Software Foundation, "PEP 20 - The Zen of Python," 2015. [Online]. Available: https://www.python.org/dev/peps/pep-0020/.

[7] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

[8] V. Frittelli, D. Serrano, R. Teicher, F. Steffolani, M. Tartabini, J. Fenández and G. Bett, "Uso de Python como Lenguaje Inicial en Asignaturas de Programación," in *Libro de Artículos Presentados en la III Jornada de Enseñanza de la Ingeniería - JEIN 2013*, Bahía Blanca, 2013.