

---

# Introducción al desarrollo de software

*Tecnicatura Universitaria en Software Libre*

Emiliano P. López  
Maximiliano Boscovich

Mayo de 2015

UNIVERSIDAD NACIONAL DEL LITORAL  
Facultad de Ingeniería y Ciencias Hídricas



*A todos los que están mirando,  
por el amor*



<b>1. Introducción</b>	<b>3</b>
1.1. Motivación	3
1.1.1. ¿Por qué Python?	3
1.2. Instalando Python	4
1.2.1. Windows	4
1.2.2. GNU/Linux	4
1.3. Entornos de programación	5
1.3.1. El intérprete interactivo	5
1.3.2. IPython, el intérprete interactivo mejorado	5
1.3.3. Entorno integrado de desarrollo (IDE)	6
1.4. Algoritmos computacionales	6
1.4.1. El primer programa	7
1.5. Modos de ejecutar tus programas	7
1.5.1. Desde la línea de comandos	7
1.5.2. Como un script	8
1.6. Elementos de un programa	8
1.6.1. Números y expresiones	8
1.6.2. Cadenas de caracteres	9
1.6.3. Comentarios	10
1.6.4. Variables	10
1.6.5. Lectura de datos	11
1.6.6. Escritura de datos	13
1.6.7. Operadores relacionales y lógicos	15
1.6.8. Funciones	15
1.6.9. Módulos	16
1.7. Ejercicios	16
<b>2. Tipos básicos</b>	<b>19</b>
2.1. Números	19
2.1.1. Reales	19
2.1.2. Complejos	20
2.2. Cadenas de texto	20
2.3. Booleanos	21
2.3.1. Operadores relacionales	21

2.3.2.	Operadores lógicos . . . . .	23
<b>3.</b>	<b>Estructuras de datos y control de flujo</b>	<b>27</b>
3.1.	Estructuras condicionales . . . . .	27
3.1.1.	Sentencia if . . . . .	27
3.1.2.	Sentencia if..else . . . . .	28
3.1.3.	Estructuras anidadas . . . . .	29
3.2.	Estructuras repetitivas . . . . .	30
3.2.1.	Estructura repetitiva <i>while</i> . . . . .	30
3.2.2.	Estructura repetitiva <i>for</i> . . . . .	30
3.3.	Estructura de datos <i>listas y diccionarios</i> . . . . .	30
3.3.1.	Listas . . . . .	30
3.3.2.	Diccionarios . . . . .	30
3.4.	Manipulando textos con strings . . . . .	30
<b>4.</b>	<b>Funciones, archivos, diccionarios</b>	<b>31</b>
4.1.	Definiendo funciones . . . . .	31
4.1.1.	Variables globales y locales . . . . .	31
4.2.	Números aleatorios . . . . .	31
4.3.	Lectura y escritura de archivos . . . . .	31
4.4.	Diccionarios . . . . .	31
<b>5.</b>	<b>Clases y objetos</b>	<b>33</b>
<b>6.</b>	<b>Indices and tables</b>	<b>35</b>

Contents:

Tabla de Contenidos

```
%%javascript  
$.getScript('https://kmahelona.github.io/ipython_notebook_goodies/ipython_notebo
```

```
<IPython.core.display.Javascript object>
```





---

# Introducción

---

En el presente capítulo introduciremos los conceptos necesarios para desarrollar los primeros algoritmos computacionales. Además, se explican las herramientas necesarias para llevar a cabo el desarrollo y sus diferentes alternativas.

## 1.1 Motivación

Gran parte de las tecnologías utilizadas en la actualidad tienen algo en común, y es que por lo general basan su lógica en algún tipo de programa. Saber programar nos permite comprender su funcionamiento y con esto nos abre un gran abanico de posibilidades, limitadas únicamente por nuestra imaginación. Pensemos por un momento en todas las aplicaciones que usamos a diario en el teléfono celular, en la PC, en la tablet, etc. Saber que si necesitamos algo en concreto seremos capaces de crearlo nosotros mismos es pura libertad.

Lo más importante de todo, es que no es necesario ser un genio para poder programar, simplemente tenemos que aprender un conjunto de reglas básicas, saber como aplicarlas y tener muchas ganas de crear cosas nuevas. Además, programar es muy divertido, al contrario de lo que mucha gente podría pensar en un principio. Es como un gran rompecabezas en el que debemos encajar ciertas piezas de una forma específica para conseguir el resultado deseado.

A lo largo de esta materia utilizaremos como lenguaje de programación a Python (<http://www.python.org>).

### 1.1.1 ¿Por qué Python?

Python es un lenguaje de programación multipropósito, poderoso y fácil

de aprender. Es del tipo interpretado, lo que quiere decir que los programas realizados con python no necesitan ser compilados, en su lugar, simplemente requieren que el equipo donde van a ser ejecutados cuente con un interprete de python instalado. Es un lenguaje que cuenta con estructuras de datos eficientes y de alto nivel. Su elegante sintaxis y su tipado dinámico hacen de éste un lenguaje ideal para el desarrollo rápido de aplicaciones en diversas áreas como ser:

- \* Aplicaciones WEB
- \* Aplicaciones científicas
- \* Gráficas
- \* Multimedia
- \* Juegos
- \* Etc.

Otra de las grandes virtudes de python, es que su interprete puede ejecutarse en la mayoría de los sistemas operativos utilizados en la actualidad (GNU/Linux, Microsoft Windows, Mac OSX, etc.).

Dada su versatilidad y simplicidad, Python es utilizado por compañías como Google, Youtube, Netflix, Yahoo, NSA, NASA, Canonical, IBM, entre otras tantas.

## 1.2 Instalando Python

Actualmente existen dos versiones de Python comúnmente utilizadas, la versión 2 y 3, ambas son completamente funcionales y muy utilizadas. En este curso nos basaremos en la versión 3.

**Ver como funcionaría miniconda en windows y linux:**  
<http://conda.pydata.org/miniconda.html>

### 1.2.1 Windows

Para instalar Python en una máquina con Windows, debemos seguir los siguientes pasos:

- Apuntar el navegador a: <https://www.python.org/downloads/windows/>
- Ir al link de la última versión disponible (por ej: latest python 3 relase)
- En la sección Files, descargar el instalador correspondiente a su arquitectura (64/32 bits), por ej: <https://www.python.org/ftp/python/3.4.3/python-3.4.3.msi>
- Ejecutar el instalador (por ej: python-3.4.3.msi) aceptando las opciones por defecto

### 1.2.2 GNU/Linux

En la mayoría de las distribuciones GNU/Linux, es muy probable que ya contemos con el intérprete instalado, incluso en sus dos versiones. En caso de no ser así, para instalarlo utilizando los administradores de paquetes debemos ejecutar los siguientes comandos desde una terminal:

Para sistemas basados en Debian (como Ubuntu o sus derivados):

```
sudo apt-get install python3
```

Para sistemas que utilizan yum como sistema de paquetes (Fedora, CentOS, RedHat)

```
sudo yum install *python*
```

## 1.3 Entornos de programación

### 1.3.1 El intérprete interactivo

Ya con el intérprete de Python instalado, podemos comenzar a programar. Si ejecutamos en una terminal `python3`, ingresaremos al intérprete en modo interactivo y veremos una salida similar a la siguiente:

```
Python 3.4.2 (default, Oct  8 2014, 10:45:20)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Con esto, el interprete de python esta listo para empezar a interpretar las instrucciones (las cuales llamaremos sentencias) que forman parte de nuestro programa, por lo que podemos decir que ya estamos listos para empezar a programar. Pero vayamos de lo más sencillo a lo más complejo, y lo mejor para comenzar es realizando ciertos cálculos matemáticos sencillos, y corroborando su resultado. Por ejemplo, escribamos lo siguiente:

```
>>> 2*5
10
>>>
```

Como vemos, si ingresamos `2*5`, le estamos diciendo al interprete de python que debe realizar la multiplicación entre 2 y 5. El interprete analiza la instrucción ingresada (`2*5`), y contesta con el resultado (10 en este caso).

Hagamos otros calculos para entrar en calor

```
>>> 2*5+10
20
>>> -3*19+3.1415
-53.8585
>>> 2/10.0
0.2
>>>
```

### 1.3.2 IPython, el intérprete interactivo mejorado

**IPython**<sup>1</sup> es una interfaz mejorada del intérprete nativo. Se lo puede utilizar en modo consola o a través de una interfaz web. La instalación en sistemas basados en Debian GNU/Linux es similar a la de python: `apt-get install ipython3`.

La ejecución de `ipython` desde una terminal nos arroja una pantalla similar a la siguiente:

```
emiliano@pynandi:~ $ ipython3
Python 3.4.2 (default, Oct  8 2014, 10:45:20)
Type "copyright", "credits" or "license" for more information.
```

---

<sup>1</sup><http://ipython.org>


```
IPython 2.3.0 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref  -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.

In [1]:
```

Otra alternativa muy interesante son los notebooks de ipython, una interfaz que permite programar utilizando el navegador web como entorno. No entraremos en detalle ya que posteriormente analizaremos su funcionamiento. Se debe ejecutar en una terminal `ipython3 notebook` y esto abrirá el navegador por defecto con el entorno cargado.

### 1.3.3 Entorno integrado de desarrollo (IDE)

Un IDE es un entorno que nos facilita las tareas a la hora de programar. Consiste en la integración de un editor de texto con características de resaltado de sintaxis, autocompletado -entre otras-, y el intérprete de Python. Existen cientos de entornos muy buenos, como por ejemplo [Spyder<sup>2</sup>](#), [PyCharm<sup>3</sup>](#) o [Ninja-IDE<sup>4</sup>](#). Para el presente curso, nos basaremos en Ninja-IDE, software libre que ha sido desarrollado por la comunidad de Python Argentina, [PyAr<sup>5</sup>](#).



files/img/u1/ninja-ide.png

Una lista bastante completa sobre las IDEs disponibles pueden encontrarse en la [wiki oficial de Python<sup>6</sup>](#)

## 1.4 Algoritmos computacionales

En forma simplificada, un programa o software es un conjunto de instrucciones que la computadora puede ejecutar. Este procedimiento formado por un conjunto de instrucciones es lo que denominamos algoritmo computacional. Una analogía a un algoritmo computacional es una receta de cocina, por ejemplo:

```
Prender el fuego
Salar la carne
Controlar cada 5 minutos hasta que haya brasas
Poner la carne a la parrilla
```

---

<sup>2</sup><https://github.com/spyder-ide/spyder>

<sup>3</sup><https://www.jetbrains.com/pycharm>

<sup>4</sup><http://ninja-ide.org>

<sup>5</sup><http://python.org.ar>

<sup>6</sup><https://wiki.python.org/moin/IntegratedDevelopmentEnvironments>

```
Cocinar hasta que esté la carne, controlar cada 5 minutos
Dar vuelta la carne
Cocinar hasta que esté la carne, controlar cada 5 minutos
Si falta sal al probar, salar
```

En esta receta se ven una serie de instrucciones que deben ser seguidas en un determinado orden, en algunos casos contamos con ingredientes, instrucciones, decisiones y acciones que se repiten. No muy distinto a un programa de computación, comencemos con algunos *ingredientes* simples de Python y veamos lo que podemos hacer con ellos.

### 1.4.1 El primer programa

El acercamiento inicial a un lenguaje de programación suele ser con el archiconocido programa “Hola mundo”. Consiste simplemente en un programa que muestra en pantalla ese mensaje.

Renunciando a cualquier pretensión de originalidad comenzaremos del mismo modo, pero despidiéndonos. Para esto utilizaremos la instrucción `print()` pasando el mensaje de despedida entre comillas, a continuación la instrucción.

```
print("Adios mundo cruel!")
```

Podemos probar la instrucción directamente desde el intérprete, creando con un editor de texto plano un archivo guardado como `chau.py` y luego ejecutándolo desde la terminal haciendo `python3 chau.py`, o bien utilizando un IDE y haciendo todo desde ahí mismo.

Ahora bien, es muchísimo más lo que podemos hacer programando además de saludar cordialmente. Veamos los elementos de un programa que nos permitirán realizar tareas más complejas y entretenidas.

## 1.5 Modos de ejecutar tus programas

El intérprete interactivo de Python es una gran ayuda para realizar pruebas y experimentar en tiempo real sobre el lenguaje. Sin embargo, cuando cerramos el intérprete perdemos lo escrito, por lo que no es una solución para escribir programas mas largos y con mayores complejidades. Por otro lado, tampoco resulta poco práctico abrir el IDE para correr un script Python. Entonces, para un programa guardado con el nombre `hola_mundo.py`, lo podemos ejecutar de las siguientes maneras:

### 1.5.1 Desde la línea de comandos

Abriendo una terminal, e invocando al intérprete python y luego la ruta y nombre del archivo:

```
$python3 hola_mundo.py
```

### 1.5.2 Como un script

Es posible ejecutarlo sin invocar al intérprete desde la línea de comandos, para esto, se debe incluir al principio del programa la siguiente línea:

```
#!/usr/bin/env python3
```

Con esa línea, estaremos especificando en el mismo programa la ruta del intérprete que debe ejecutarlo. Antes de poder ejecutarlo, debemos otorgarle permisos de ejecución con el comando del sistema operativo `chmod`:

```
$chmod +x hola_mundo.py
```

Una vez realizado lo anterior, es posible ejecutarlo desde la terminal, como cualquier ejecutable del sistema operativo, llamándolo con el nombre del programa antecediendo `./` (punto barra, sin comillas):

```
$./hola_mundo.py  
Adiós mundo cruel
```

## 1.6 Elementos de un programa

A continuación veremos los ingredientes fundamentales de un lenguaje de programación como Python, para llevar a cabo los ejemplos utilizaremos el intérprete interactivo mejorado `ipython`.

### 1.6.1 Números y expresiones

Frecuentemente requerimos resolver cálculos matemáticos, las operaciones aritméticas básicas son:

- adición: `+`
- sustracción: `-`
- multiplicación: `*`
- división: `/`
- módulo: `%`
- potencia: `**`
- división entera: `//`

Las operaciones se pueden agrupar con paréntesis y tienen precedencia estándar. Veamos unos ejemplos.

```
In [9]: 1/3  
Out[9]: 0.3333333333333333  
  
In [10]: 1//3
```

```
Out[10]: 0

In [11]: 10%3
Out[11]: 1

In [12]: 4%2
Out[12]: 0
```

El caso de la potencia, también nos sirve para calcular raíces. Veamos una potencia al cubo y luego una raíz cuadrada, equivalente a una potencia a la 1/2.

```
In [13]: 5**3
Out[13]: 125

In [14]: 2**(1/2)
Out[14]: 1.4142135623730951
```

Los datos numéricos obtenidos en las operaciones previas se clasifican en reales y enteros, en python se los clasifica como float e int respectivamente, además existe el tipo complex, para números complejos.

Utilizando la función type() podemos identificar el tipo de dato. Veamos:

```
In [15]: type(0.333)
Out[15]: float

In [16]: type(4)
Out[16]: int
```

## 1.6.2 Cadenas de caracteres

Además de números, es posible manipular texto. Las cadenas son secuencias de caracteres encerradas en comillas simples ('...') o dobles ("..."), el tipo de datos es denominado *str* (string). Sin adentrarnos en detalles, que posteriormente veremos, aquí trataremos lo indispensable para poder desarrollar los primeros programas. Veamos unos ejemplos:

```
>>> 'huevos y pan'          # comillas simples
'huevos y pan'
```

Los operadores algebraicos para la suma y multiplicación tienen efecto sobre las cadenas:

```
>>> 'eco '*4                # La multiplicación repite la cadena
'eco eco eco eco '

>>> 'yo y ' + 'mi otro yo'  # La suma concatena dos o mas cadenas
'yo y mi otro yo'
```

Es posible utilizar cadenas de más de una línea, anteponiendo **triples comillas** simples o dobles al inicio y al final, por ejemplo (fragmento del poema de Fortunato Ramos *Yo jamás fui un niño*):

```
'''  
Mi sonrisa es seca y mi rostro es serio,  
mis espaldas anchas, mis músculos duros  
mis manos partidas por el crudo frío  
sólo ocho años tengo, pero no soy un niño.  
'''
```

### 1.6.3 Comentarios en el código

En los ejemplos previos y siguientes, veremos dentro del código comentarios explicativos que no serán ejecutados por el intérprete. Su uso solamente está destinado a quien lea el código, como texto explicativo para orientar sobre lo que se realiza.

Los comentarios pueden ser de una única o múltiples líneas. Para el primer caso se utiliza el símbolo numeral. Lo que continúa a la derecha de su uso no es ejecutado.

Los comentarios de múltiples líneas se deben escribir entre triples comillas, ya sean simples o dobles.

### 1.6.4 Variables

Las variables son contenedores para almacenar información. Por ejemplo, para elevar un número al cubo podemos utilizar 3 variables, para la base (*num1*), para el exponente (*num2*) y para almacenar el *resultado*:

```
num1 = 5                # num1 toma valor 5.  
num2 = 3                # num2 toma 3.  
resultado = num1**num2  # resultado toma num1 elevado a num2.  
print('El resultado es', resultado)
```

El operador igual (=) sirve para asignar lo que está a su derecha, a la variable que se encuentra a su izquierda. Implementemos la siguiente ecuación para dos valores de  $x$ , 0.1 y 0.2.

$$y = (x - 4)^2 - 3$$

```
x1 = 0.1  
y1 = (x1-4)**2-3  
  
x2 = 0.2  
y2 = (x2-4)**2-3  
  
print(x1, y1)  
print(x2, y2)
```

Veremos la siguiente salida por pantalla:



```
0.1 12.209999999999999
0.2 11.44
```

Otros ejemplos utilizando variables que contengan **cadenas de caracteres**:

```
cadena1 = 'siento que '
cadena2 = 'nací en el viento '

cadena3 = cadena1 + cadena2

print(cadena3)
```

Los nombres de las variables (identificador o etiqueta) pueden estar formados por letras, dígitos y guiones bajos, teniendo en cuenta ciertas restricciones, no pueden comenzar con un número y ni ser algunas de las siguientes palabras reservadas:

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Se debe tener en cuenta que las variables diferencian entre mayúsculas y minúsculas, de modo que juana, JUANA, JuAnA, JUANa son variables diferentes. Esta característica suele denominarse como *case-sensitive*.

### 1.6.5 Lectura de datos

De los ejemplos que vimos, los valores que almacenan las variables fueron ingresados en el mismo código, difícilmente sea útil contar con los valores cargados en el programa en forma estática. Por esta razón, generalmente se requiere leer información de diferentes fuentes, puede ser desde un archivo o bien interactuando con un usuario.

La lectura de datos desde el teclado se realiza utilizando la sentencia *input()* del siguiente modo:

```
nombre = input("¿Cómo es su nombre, maestro? ")
print("Hola, " + nombre + "!")
```

El comportamiento es:

```
¿Cómo es su nombre, maestro?
Juan de los palotes
Hola, Juan de los palotes!
```

Es importante tener en cuenta que toda lectura por teclado utilizando la función *input()* va a almacenar lo ingresado como una variable de tipo *str*, es decir una cadena de caracteres. Veamos el comportamiento al sumar dos números:

```
num1 = input("Ingrese un número = ")
num2 = input("Ingrese otro número = ")
print("El resultado es =", num1+num2)
```

```
Ingrese un número = 28
Ingrese otro número = 03
El resultado es = 2803
```

Claramente la suma de los valores ingresados no da el resultado observado. El inconveniente se debe a que ambos valores son tomados como cadenas de caracteres y la operación de suma entre cadenas de caracteres produce la concatenación de las mismas. Es necesaria convertir la cadena de caracteres (str) a un valor numérico, ya sea entero o real (int o float).

Para convertir datos de diferentes tipo se utilizan las funciones int(), float() o str(). Modificando el caso anterior:

```
num1 = int(input("Ingrese un número = "))
num2 = int(input("Ingrese otro número = "))
print("El resultado es =", num1+num2)
```

```
Ingrese un número = 28
Ingrese otro número = 03
El resultado es = 31
```

Veamos un ejemplo para operar directamente el valor leído en una ecuación matemática con el siguiente código:

```
x = input("Ingrese x = ")
y = (x-4)**2-3
print(x, y)
```

```
Ingrese x = 3
```

```
-----
TypeError                                Traceback (most recent call last)

<ipython-input-3-3baa5c95d16e> in <module>()
      1 x = input("Ingrese x = ")
----> 2 y = (x-4)**2-3
      3 print(x, y)

TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

A diferencia del ejemplo visto anteriormente, donde la suma de dos cadenas era una operación perfectamente válida, ahora nos encontramos con operaciones entre diferentes tipos pero incompatibles. En este caso, podemos convertir la entrada en un número flotante para operar con normalidad:

```
x = float(input("Ingrese x = "))
y = (x-4)**2-3
print(x, y)
```

```
Ingrese x = 3
3.0 -2.0
```

Es posible combinar distintos tipos de datos haciendo la conversión correspondiente, en el último ejemplo, tanto  $x$  como  $y$  son de tipo *float* y es posible concatenarlos a una cadena de caracteres haciendo la conversión correspondiente, utilizando la función *str()*:

```
mensaje = 'y vale ' + str(y) + ' para un valor de x = ' + str(x)
```

### 1.6.6 Escritura de datos

Hemos hecho uso de la función *print()* en su mínima expresión. Iremos viendo diferentes usos a partir de las siguientes variables:

```
# Variables a imprimir
cad = 'Pi es'
pi = 3.1415
mil = 1000
uno = 1
```

#### Como argumentos

La forma más simple es separar los argumentos a ser impresos mediante comas.

```
print(cad, pi, 'aproximadamente')
```

```
Pi es 3.1415 aproximadamente
```

Por defecto, la separación que se obtiene entre cada argumento es un espacio en blanco, sin embargo, se puede cambiar este comportamiento agregando como argumento *\*sep=' '\** y entre las comillas incluir el separador deseado, por ejemplo:

```
print(cad, pi, 'aproximadamente', sep=';')
print(cad, pi, 'aproximadamente', sep=',')
print(cad, pi, 'aproximadamente', sep=':-)')
```

```
Pi es;3.1415;aproximadamente
Pi es,3.1415,aproximadamente
Pi es:-)3.1415:-)aproximadamente
```

Como vemos, en cada ejecución la impresión se realiza en diferentes renglones, este es el comportamiento por defecto, que puede ser modificando agregando el parámetro *\*end=""*. Reflejemos esto con un ejemplo:

```
print(1, end=" ")
print(2, end=" ")
print(3)
print(4)
```

```
1 2 3
4
```

### Usando comodines

Los comodines consisten en una marca especial en la cadena a imprimir que es reemplazada por la variable y el formato que se le indique. Existen tres tipos de comodines, para números enteros, reales (flotantes) y para cadenas de caracteres:

- Comodín para reales: %f
- Comodín para enteros: %d
- Comodín para cadenas: %s

Se utilizan del siguiente modo:

```
print('Pi es %f aproximadamente' %pi)
print('El número %d es %s que %d' %(mil, "menor", mil-1))
```

```
Pi es 3.141500 aproximadamente
El número 1000 es menor que 999
```

Es posible formatear los valores, elegir el ancho del campo, la cantidad de decimales, entre muchas otras funciones.

```
print('% .2f % .4f % .3f' %(pi,pi,pi))
print('%4d' %uno)
```

```
-----
NameError                                Traceback (most recent call last)

<ipython-input-1-f45a2755e54d> in <module>()
----> 1 print('% .2f % .4f % .3f' %(pi,pi,pi))
      2 print('%4d' %uno)

NameError: name 'pi' is not defined
```

La sintaxis general del uso de comodines es:

```
%[opciones][ancho][.precisión]tipo
```

Algunas variantes de lo visto se explica en la siguiente lista:

- %d : un entero
- %5d: un entero escrito en un campo de 5 caracteres, alineado a la derecha
- %-5d: un entero escrito en un campo de 5 caracteres, alineado a la izquierda
- %05d: un entero escrito en un campo de 5 caracteres, completado con ceros desde la izquierda (ej. 00041)

- %e: flotante escrito en notación científica
- %E: como %e, pero E en mayúscula
- %11.3e: flotante escrito en notación científica con 3 decimales en un campo de 11 caracteres
- %.3e: flotante escrito en notación científica con 3 decimales en un campo de ancho mínimo
- %5.1f: flotante con un decimal en un campo de 5 de caracteres
- %.3f: flotante con 3 decimales en un campo de mínimo ancho
- %s: una cadena
- %-20s: una cadena alineada a la izquierda en un campo de 20 caracteres de ancho

Con lo visto hasta aquí tenemos suficientes alternativas para mostrar en pantalla información de diferentes tipos. Existen una alternativa para imprimir en pantalla utilizando el método `format`, el lector interesado puede indagar más al respecto en <http://docs.python.org.ar/tutorial/3/inputoutput.html>, en el capítulo Entrada y Salida del tutorial de Python oficial <http://docs.python.org.ar/tutorial/pdfs/TutorialPython3.pdf> ó también en [http://www.python-course.eu/python3\\_formatted\\_output.php](http://www.python-course.eu/python3_formatted_output.php)

## 1.6.7 Funciones

Las funciones son programas o subprogramas que realizan una determinada acción y pueden ser invocados desde otro programa. En los capítulos posteriores trabajaremos intensamente con funciones creando propias, sin embargo en esta sección, con el fin de comprender su uso, presentaremos algunas pocas de las que nos provee Python.

El uso de funciones nativas en Python es directo, veamos algunas:

```
frase = 'simple es mejor que complejo'
num_letras = len(frase)
print(num_letras)
```

28

El ejemplo previo hicimos uso de dos funciones, por un lado la función `*print()*`, presentada ya desde el primer programa y una nueva función, `*len()*`, que recibe como dato de entrada una cadena de caracteres y calcula la cantidad de caracteres de la misma y lo retorna de manera tal que lo podemos asignar a una variable (`num_letras`).

ver ejemplo 2-3 pág 38 de Beginning Python from novice to professional 2nd edition.

Estos subprogramas pueden estar vinculadas, de modo que se organizan en módulos.

## 1.6.8 Módulos

Python posee cientos de funciones que se organizan o agrupan en módulos. Veamos un ejemplo para calcular la raíz cuadrada, el seno y coseno de un número haciendo uso de las funciones

*sqrt()*, *sin()* y *cos()*, todas ubicadas bajo el módulo *math*.

```
import math

nro = 2
raiz = math.sqrt(nro)
print("La raiz de %d es %.4f" %(nro, raiz))
print("El seno de %d es %.4f" %(nro, math.sin(nro)))
print("El coseno de %d es %.4f" %(nro, math.cos(nro)))
```

```
La raiz de 2 es 1.4142
El seno de 2 es 0.9093
El coseno de 2 es -0.4161
```

Del ejemplo previo, hemos visto como indicarle a Python que importe -o haga uso de- un módulo en particular y de algunas de sus funciones incluidas.

En capítulos posteriores veremos en profundidad distintos modos de importar módulos e invocar sus funciones.

---

### Ejercicios

---

1- Realice un programa que permita al usuario ingresar una temperatura en grados centígrados y que muestre su equivalente en grados fahrenheit.

```
Ingrese temperatura en °C: 33.8
Conversión a Fahrenheit: 92.84
```

2- Realice un programa que permita al usuario ingresar su nombre y que luego lo muestre repetido en pantalla tantas veces como cantidad de letras posea el nombre.

3- Ingrese el nombre y edad de dos personas en variables separadas (nom1, edad1, nom2, edad2). Luego, intercambie la edad y muestre el resultado en pantalla. Indague de qué manera puede intercambiar el contenido de variables en Python.

4- La simple tarea de realizar la cocción de un huevo pasado por agua tiene sus secretos. Con la ecuación a continuación se puede conocer el tiempo en alcanzar el punto exacto. Programe la ecuación para valores de bla bla bla

5- Lea por teclado el valor del cuenta kilómetros de un automovil, posteriormente, permita ingresar el nuevo valor luego de realizar un viaje y muestre en pantalla los kilómetros recorridos, así como también ese valor en metros, centímetros, yardas y pies.

6- Las benévolas compañías telefónicas cobran la tarifa de cada llamada del siguiente modo: un valor fijo de \$0.80 cuando se establece la llamada, luego, fracciona por tiempo, donde el primer minuto tiene un valor de \$1.30 y los subsiguientes de \$1.45. Realice un programa que permita ingresar la duración de una llamada y que muestre luego el costo total de la misma, a la que se le debe agregar un porcentaje del 20 % correspondiente a impuestos.

7- Un atleta realiza sus entrenamientos para una maratón (42.195km) y desea conocer su velocidad promedio. Desarrolle un programa donde se ingrese el tiempo transcurrido en tres variables diferentes: horas, minutos y segundos. Luego, muestre la velocidad promedio en km/h y km/seg.

8- En el siguiente programa se calcula la diferencia de tiempo entre dos marcas de tiempo. Analice el código del programa y explique las acciones que se llevan a cabo. Luego, modifíquelo para que las dos marcas de tiempo sean ingresadas por un usuario.

```
# dos marcas de tiempo
hora1,min1,seg1 = 14, 58, 59
hora2,min2,seg2 = 16, 0, 0
```

```
# conversión del tiempo a segundos
t1s = hora1*60*60 + min1*60 + seg1
t2s = hora2*60*60 + min2*60 + seg2

# diferencia
t = abs(t1s-t2s)

# cálculos de hora, minuto y segundos
h = t//3600
m = (t-h*3600)//60
s = t-h*3600-m*60

# impresión en pantalla
print ('Diferencia de tiempo:', h, 'hs', m, 'min', s, 'seg')
```

### Tabla de Contenidos

```
%%javascript
$.getScript('https://kmahelona.github.io/ipython_notebook_goodies/ipython_notebo
```

```
<IPython.core.display.Javascript object>
```



---

## Tipos básicos

---

Como vimos en la Unidad 1, las variables pueden contener diferentes tipos de datos, y al ser distintos, son tratados de manera diferente por Python (por ejemplo no podemos sumar un número con una letra).

Hemos visto 2 de los 3 tipos básicos que utiliza python, los cuales se dividen en: \* **Números** \* **Cadenas de texto** \* **Booleanos**

### 3.1 Números

Los números como vimos pueden ser enteros, reales (también denominados de coma flotante) ó complejos. ### Enteros Los números enteros son aquellos números positivos o negativos que no tienen decimales (además del cero). En Python se representan mediante el tipo int (de integer, entero). Por ejemplo:

```
a = 4
type(a)
```

```
int
```

#### 3.1.1 Reales

Los números reales son los que tienen decimales. En Python se expresan mediante el tipo float. Para representar un número real en Python se escribe primero la parte entera, seguido de un punto y por último la parte decimal. Por ejemplo:

```
real = 6.2231
```

También se puede utilizar notación científica, y añadir una e (de exponente) para indicar un exponente en base 10. Por ejemplo:

```
real = 0.6e-3
```

Lo que sería equivalente a  $0.6 \times 10^{-3} = 0.6 \times 0.001 = 0.0006$

```
real = 8.21
type(real)
```

```
float
```

### 3.1.2 Complejos

Los números complejos son aquellos que tienen parte imaginaria. Si no conocías de su existencia, es más que probable que nunca lo vayas a necesitar, de hecho la mayor parte de los lenguajes de programación carecen de este tipo, aunque sea muy utilizado por ingenieros y científicos en general.

En el caso de que necesites utilizar números complejos, debes saber que son llamados `complex` en Python, y que se representan de la siguiente forma:

```
c = 4 + 5j
type(c)
```

```
complex
```

## 3.2 Cadenas de texto

Tal como hemos visto en la unidad anterior, las cadenas (`string` en inglés ó `str`) no son más que texto encerrado entre comillas simples (`'cadena'`), dobles (`"cadena"`) ó triples(`'''Cadenas multilíneas'''`). Por ejemplo:

```
a = 'El futuro mostrará los resultados y juzgará a cada uno de acuerdo a sus logros'
type(a)
```

```
str
```

```
b = "En realidad no me preocupa que quieran robar mis ideas, me preocupa que ellos no lo hagan"
type(b)
```

```
str
```

```
c = '''Un instrumento de poco costo y no más grande que un reloj, permitirá a su dueño, en cualquier parte, ya sea en el mar o en la tierra, música, canciones o un discurso dictado en cualquier otro sitio distante. Del mismo modo, cualquier dibujo o impresión transferida de un lugar a otro (Nikola Tesla, ~ año 1891).'''
type(c)
```

```
str
```

### 3.3 Booleanos

Por último, nos queda el tipo básico booleano. Una variable de tipo booleano sólo puede tener dos valores: True (cierto) y False (falso). Estos valores son especialmente importantes para las expresiones condicionales y los bucles, como veremos más adelante. Pero veamos algunos ejemplos:

```
a = True
type(a)
```

```
bool
```

```
b = False
type(b)
```

```
bool
```

```
c = 10 > 2
print (c)
```

```
True
```

En este último ejemplo vemos algo particular, hemos asignado a la variable **c** el resultado de una expresión lógica ( $10 > 2$ ). Python en este caso opera con la misma y asigna a la variable **c** el resultado de dicha operación, la cual en este caso es verdadera (True), dado que 10 es mayor que 2. Al tratarse se una operación lógica, el resultado siempre será de tipo booleano (bool), es decir, será verdadero o será falso.

```
type(c)
```

```
bool
```

#### 3.3.1 Operadores relacionales

Como vimos en el ejemplo anterior, los valores booleanos son además el resultado de expresiones que utilizan operadores relacionales (comparaciones entre valores).

Estos operadores, siempre se utilizan de la siguiente manera:

operando\_A (operador) operando\_B

Por ejemplo:

```
10 > 4
```

```
True
```

En este caso el operando A es 10 y el B es 4, el resultado de aplicar el operador “>” a los operandos A y B en este caso es True (cierto) dado que 10 es mayor que 4.

La lista completa de operadores que podemos utilizar en python es:

Operador	Descripción	Ejemplo	Resultado
==	¿son iguales a y b?	5 == 3	False
!=	¿son distintos a y b?	5 != 3	True
<	¿es a menor que b?	5 < 3	False
>	¿es a mayor que b?	5 > 3	True

Veamos otro ejemplo, ahora con cadenas de texto:

```
d = "Una cosa" == "Otra cosa"
print (d)
```

```
False
```

En este caso el operador == se utiliza para comparar si son iguales los operandos. Esta comparación se hace caracter a caracter, por lo que al ser diferentes las cadenas, el resultado es False. Lo siguiente también es False

```
d = "Una cosa" == "una cosa"
print (d)
```

```
False
```

Solo cuando ambas cadenas son iguales, la comparación devuelve verdadero

```
d = "Una cosa" == "Una cosa"
print (d)
```

```
True
```

El tipo como hemos visto, es booleano:

```
type(d)
```

```
bool
```

También podemos comparar números, expresiones algebraicas y expresiones lógicas.

```
resultado = 24 > 3*7
print (resultado)
```

```
True
```

```
resultado = False == True
print (resultado)
```

```
False
```

```
a = 2*8
b = 3*8
resultado = (a < b)
print (resultado)
```

```
True
```

En Python, una expresión que es cierta tiene el valor 1, y una expresión que es falsa tiene el valor 0.

```
a = True
resultado = a == 1
print (resultado)
```

```
True
```

```
b = False
resultado = b == 0
print (resultado)
```

```
True
```

### 3.3.2 Operadores lógicos

Además de los operadores relacionales, tenemos los operadores lógicos. Existen 3 tipos de operadores lógicos: **\*\*and (y), or (ó), y not (no)\*\***. Por ejemplo:

$x > 0$  **\*\*and\*\***  $x < 10$

es verdadero sólo si  $x$  es mayor que 0 **\*\*y\*\*** menor que 10.

$n \% 2 == 0$  **\*\*or\*\***  $n \% 3 == 0$

es verdadero si cualquiera de las condiciones es verdadera, o sea, si el número es divisible por 2 o por 3. O sea, podemos leer la línea anterior como  *$n$  dividido 2 es igual a 0* **\*\*ó\*\***  *$n$  dividido 3 es igual a 0*.

Finalmente, el operador **not** niega una expresión booleana, de forma que

**\*\*not\*\***( $x > y$ ) es cierto si ( $x > y$ ) es falso, o sea, si  $x$  es menor o igual que  $y$ .

En resumen tenemos los siguientes operadores lógicos

Operador	Descripción	Ejemplo	Resultado
<b>and</b>	¿se cumple a y b?	True <b>and</b> False	False
<b>or</b>	¿se cumple a o b?	True <b>or</b> False	True
<b>not</b>	No a	<b>not</b> True	False

Veamos algunos ejemplos

```
a = 9
b = 16
c = 6
resultado = (a < b) and (a > c)
print (resultado)
```

```
True
```

En este caso, como ambas operaciones devuelven True (verdadero), el resultado es verdadero.

```
a = 9
b = 16
c = 6
resultado = (a < b) and (a < c)
print (resultado)
```

```
False
```

Por el contrario, si una de las condiciones devuelve False, el resultado será False.

Veamos algunos ejemplos con el operador **\*or\***

```
a = 9
b = 16
c = 6
resultado = (a < b) or (a < c)
print (resultado)
```

```
True
```

En este caso la primer operación es verdadera y la segunda es falsa, pero como estamos utilizando el operador **\*or\***, la variable resultado tendrá como valor True.

Por último, veamos un ejemplo con el operador **\*not\***

```
a = 9
b = 16
resultado = not(a > b)
print (resultado)
```

```
True
```

En este ejemplo *a* es menor que *b*, por lo que la expresión es falsa. Sin embargo al utilizarse el operador **\*not\*** estamos cambiando el resultado por su opuesto (en este caso True). La expresión podría leer como “no es cierto que *a* es mayor que *b*”, lo cual es una expresión cierta, y por lo tanto el valor correspondiente es True.

Veamos un ejemplo un poco mas complicado

```
a = 9
b = 16
resultado = (not(a > b)) and (not(b < c))
print (resultado)
```

```
True
```

Desgloceemos un poco este ejemplo:

En este caso la expresión (*a* > *b*) es falsa, al igual que (*b* < *c*), por lo que podríamos ver a lo anterior como

```
resultado = (not(False)) and (not(False))
```

Dijimos que el operador **\*not\*** cambia el resultado de una expresión booleana por su opuesto, por lo que si seguimos desarrollando esta línea tenemos:

```
resultado = (True) and (True)
```

Como ambas expresiones son verdaderas, el valor de la variable *resultado* será *True*.

Se debe tener un especial cuidado con el orden en que se utilizan los operadores. Para asegurarnos de que estamos aplicando los operadores a una expresión particular, siempre es recomendable utilizar paréntesis para demarcar la expresión sobre la que deseamos operar.

### Referencias utilizadas en esta unidad:

- **\*Python para todos\***, Raúl González Duque, <http://mundogeek.net/tutorial-python>

#### Tabla de Contenidos

```
%%javascript
$.getScript('https://kmahelona.github.io/ipython_notebook_goodies/ipython_notebo
```

```
<IPython.core.display.Javascript object>
```





---

## Estructuras de datos y control de flujo

---

**NOTA EMI:** Poner algo sobre las instrucciones secuenciales, donde el flujo del programa es inevitablemente único y donde el recorrido consiste en ejecutar una secuencia de pasos, sin poder cambiar o saltar algunas instrucciones, sin poder repetir etc, etc, etc... Teorema fundamental de la programación estructurada

Si un programa no fuera más que una lista de órdenes a ejecutar de forma secuencial, una por una, no tendría mucha utilidad. Es por ello que en la mayoría de los lenguajes de programación existen lo que se denominan estructuras de control. Estas estructuras permiten que, ante determinadas condiciones, un programa se comporte de diferentes maneras.

Supongamos que queremos hacer un programa que nos haga ciertas preguntas y en base a las respuestas determine si nos conviene ir al trabajo en bicicleta o en auto. Este programa podría considerar inicialmente la temperatura ambiente, la hora y la distancia. Estos indicadores (variables), determinarían si el programa se debe comportar de una forma u de otra para de este modo recomendarnos una cosa (el uso de la bicicleta) u otra (el auto).

### 4.1 Estructuras condicionales

La primer estructura de control que veremos son los condicionales, los cuales nos permiten comprobar condiciones y hacer que se ejecute un fragmento de código u otro, dependiendo de esta condición. Aquí es donde cobra su importancia el tipo booleano que aprendimos en la sección anterior sobre los tipos básicos.

#### 4.1.1 Sentencia *if*

La forma más simple de un estamento condicional es un **\*if\*** (del inglés si) seguido de la condición a evaluar, dos puntos (:) y en la siguiente línea e indentado(con sangría), el código a ejecutar en caso de que se cumpla dicha condición. Por ejemplo, si consideramos lo anterior, y hacemos que el programa por ahora solo considere la temperatura, podríamos hacer lo siguiente:

```
temperatura = 12
if (temperatura > 10) and (temperatura < 30):
    print('Está lindo para bici!')
```

```
Deberías ser amable con el medio ambiente e ir en bicicleta
```

Esta sentencia se lee como: si (if) temperatura mayor a 10 y menor a 30, entonces ejecutar: print('Está lindo para bici!'). Estas sentencias solo se ejecutarán si se cumple la condición de que la variable temperatura contenga un valor que este entre 10 y 29, para el caso donde temperatura sea menor a 10 o mayor a 29, el programa no hará nada.



files/img/u3/ej\_sentencia\_if.png

Una característica saliente para este tipo de comparaciones en Python es la de asemejar al lenguaje natural, por lo que podemos implementar la comparación previa haciendo:

```
if 10 < temperatura < 30:
    print('Está lindo para bici!')
```

### ¿Qué acciones se ejecutan al cumplirse la condición?

Una cuestión muy importante es indentar tal como se ha hecho en el ejemplo, es decir, asegurarnos de dejar una sangría en las líneas debajo de los 2 puntos (:) que se deben ejecutar en caso que la condición de la pregunta se cumpla.

Todo lenguaje de programación tiene en su sintaxis un modo de identificar las acciones que forman parte de un bloque, en Python esto es a partir de la sangría.

### 4.1.2 Sentencia *if..else*


Nuestro interés inicial era que el programa nos dijera si podemos ir en auto o en bicicleta, y el ejemplo anterior solo nos dice algo cuando podemos ir en bici, pero no dice o hace nada cuando la condición no se cumple. Para estos casos existe un condicional llamado *\*else\** (del inglés si no), que se usa conjuntamente con if y que sirve para ejecutar ciertas instrucciones en caso de que la condición de la sentencia if no se cumpla. Por ejemplo:

```
if (temperatura > 10) and (temperatura < 30):
    print('Está lindo para ir en bici')
else:
    print('Te recomiendo ir en cole')
```

```
Está lindo para ir en bici
```

Esto se lee como *si temperatura es mayor o igual a 10 y temperatura es menor que 30, entonces mostrar el mensaje 'Está lindo para ir en bici', sino mostrar el mensaje 'Te recomiendo ir en cole'*. Siempre se ejecutará una opción u otra, dependiendo del valor de la variable temperatura.

Por lo que en este punto podemos decir que el código se bifurca en dos caminos diferentes dependiendo de una condición (que en este caso es el valor de la variable temperatura).



files/img/u3/ej\_sentencia\_if\_else.png

En este caso también tenemos que prestar atención a la indentación utilizada. La sentencia *else* se escribe al mismo nivel que la sentencia *if*, y las sentencias que se deben ejecutar en caso de no se cumpla la condición *if*, deben ir indentadas también.

Una versión más completa del programa podría ser la siguiente:


```
temperatura = int(input('Ingrese la temperatura en °C:'))

if (temperatura > 10) and (temperatura < 30):
    print('Está lindo para ir en bici')
else:
    print('Te recomiendo ir en cole')

print('Que tenga buen día!')
```

```
Ingrese la temperatura en °C:12
Deberías ser amable con el medio ambiente e ir en bicicleta
Que tenga buen día!
```

En este caso consultamos por la temperatura, pidiéndole al usuario que la ingrese por teclado (para esto utilizamos la función *input* que vimos en la Unidad 1). Luego mostramos en pantalla lo que corresponda según el valor ingresado, y por último mostramos el mensaje ‘Que tenga buen día!’. Es importante mencionar que la última sentencia siempre se ejecutará, la bifurcación se produce solamente entre las sentencias que estan dentro del *if* y el *else*, lo restante se seguirá ejecutando de manera secuencial.



files/img/u3/ej\_sentencia\_if\_else\_completa.png

### 4.1.3 Estructura de selección múltiple *if..elif..else*

En los dos casos previos la secuencia de ejecución del programa tiene solamente dos alternativas, si la condición es verdadera (*True*) o si es falsa (*False*), incluso puede no existir un camino por la alternativa falsa, tal como se planteó en el primer ejemplo.

Las estructuras de selección múltiple sirven para evaluar mas de una condición y por ende posibilitar varios caminos de ejecución del programa. En Python, la forma de esta estructura es del siguiente modo:

```
if condicion1:
    acciones
    ...
elif condicion2:
    acciones
    ...
elif condicion3:
    acciones
    ...
else:
    acciones
    ...
```

La interpretación de esta sentencia significa que cuando cumpla alguna de las condiciones ingresará al bloque de acciones correspondientes y, en caso que no cumpla con ninguna, ejecutará las acciones del `else`, que podría ser omitido si no son necesarias acciones por defecto.

Veamos un ejemplo para mejorar la comprensión. Se lee una nota numérica de una evaluación (0..100) y el programa debe mostrar una calificación cualitativa según la siguiente escala:

- Insuficiente (nota < 60)
- Aprobado (60 <= nota < 70)
- Bueno (70 <= nota < 80)
- Muy Bueno (80 <= nota < 90)
- Distinguido (90 <= nota < 100)
- Sobresaliente (nota = 100)

```
# Lectura de la nota
nota = int(input('Ingrese la nota (0..100): '))
# Decide la calif. correspondiente
if nota < 60:
    calif = "Insuficiente"
elif 60 <= nota < 70:
    calif = "Aprobado"
elif 70 <= nota < 80:
    calif = "Bueno"
elif 80 <= nota < 90:
    calif = "Muy Bueno"
elif 90 <= nota < 100:
    calif = "Distinguido"
else:
    calif = "Sobresaliente"
# Mensaje alusivo
print("Calificación: ", calif)
```

```
Ingrese la nota (0..100): 98  
Calificación: Distinguido
```

Como se observa, cada expresión condicional planteada es excluyente de las demás, por lo que no puede cumplir con mas de una a la vez. Ahora, podría existir un planteo donde se cumplan más de una condición y la pregunta obvia es, ¿qué sucede en ese caso?

En el siguiente programa, ¿qué mensaje se muestra en pantalla?

```
val = 85  
if val > 81:  
    print("opción 1")  
elif val > 82:  
    print("opción 2")  
elif val > 83:  
    print("opción 3")
```

#### 4.1.4 Estructuras anidadas

Retomando el ejemplo del programa anterior, supongamos ahora que también queremos considerar la distancia que se debe recorrer. En este caso deberíamos preguntar por la distancia, pero también por la temperatura. Para que en los casos donde la temperatura sea agradable, la distancia no sea demasiado larga como para ir en bicicleta.

Para estos casos, se pueden utilizar estructuras anidadas, es decir, en el bloque de código que se ejecutará en caso de cumplirse o no una determina condición, podemos poner una nueva estructura de control, por ejemplo un nuevo *if*.

Reescribamos el código anterior para que considere esta nueva condición, y veamos como usar estructuras anidadas:

```
temperatura = int(input('Ingrese la temperatura en °C:'))  
distancia = int(input('Ingrese la distancia a recorrer en km:'))  
  
if (temperatura > 10) and (temperatura < 30):  
    if (distancia <= 15):  
        print('Está lindo para ir en bici')  
    else:  
        print('Está lindo, pero es lejos, le recomiendo ir en auto')  
else:  
    print('La temperatura no es agradable, le recomiendo ir en auto.')  
  
print('Que tenga buen día!')
```

```
Ingrese la temperatura en °C:15  
Ingrese la distancia a recorrer en km:1  
Está lindo para ir en bici  
Que tenga buen día!
```

En este caso si se cumple la condición de que la variable temperatura contiene un valor entre 10 y 29, se pasa a considerar el valor de la variable distancia; si esta es menor o igual a 15,

se muestra el mensaje *‘Está lindo para ir en bici’*, en caso contrario, se muestra el mensaje *‘Está lindo, pero es lejos, le recomiendo ir en auto’*. Por otro lado, si el valor de la variable temperatura no esta entre 10 y 29, se seguirá mostrando el mensaje *‘La temperatura no es agradable, le recomiendo ir en auto’*. Lo mismo sucede con la última sentencia, la cual mostrará el mensaje *‘Que tenga buen día!’* independientemente del valor de las variables *temperatura* y *distancia*

## 4.2 Estructuras repetitivas

Ahora podemos dotar a nuestros programas de mayor complejidad, combinando y anidando las estructuras condicionales vistas. Sin embargo, aún tenemos una limitante, cada instrucción tendrá vida al momento de ser ejecutada e inmediatamente después no se ejecutará más hasta que se el programa se invoque nuevamente.

Imaginemos que debemos consultar la pregunta de la temperatura a cientos de miles de personas, deberíamos ejecutar cientos de miles de veces el programa, iniciándolo y esperando su finalización para repetir el proceso una y otra vez. Se hace evidente la ausencia de una estructura que permita repetir cuantas veces se requiera una determinada acción, aquí es donde entran en acción las estructuras repetitivas.

### 4.2.1 Sentencia *while*

El *while* permite repetir una serie de acciones mientras que una determinada expresión (o condición) se cumpla, en caso contrario, se finaliza la repetición.

Una expresión se cumple cuando arroja un resultado verdadero, que en Python es `True`. La estructura del *while* es la siguiente:

```
while <expresion>
    accion1
    accion2
    ...
    accionN
```

Tal como se explicó previamente, las acciones que se repiten en cada iteración son aquellas que tienen sangría, lo que indica que son partes del ciclo *while*.

### Bucles condicionales

Veamos un ejemplo donde se le pregunte el valor de temperatura a 5 personas y sugiera ir caminando si el clima es agradable (mayor a 16°C) o en caso contrario en vehículo. Tomemos una estrategia para resolver el problema:

1. Leemos una temperatura que se ingresa por teclado
2. Escribimos en pantalla un mensaje según la temperatura
3. Repetir los dos pasos previos un total de cinco veces

## Pasos 1 y 2

```
temperatura = int(input('Ingrese la temperatura en °C:'))
if (temperatura > 16):
    print('Vas caminando')
else:
    print('Mucho frío, en vehículo')
```

## Paso 3

Debemos englobar los pasos previos en una estructura que repita 5 veces. Pensemos lo anterior como un único bloque denominado *Pasos1y2*, y una manera de controlar cinco repeticiones. Para ésto, usamos una variable con un valor inicial conocido (1) que incrementamos en una unidad luego de cada ejecución del bloque que denominamos *Pasos1y2*. La estructura de nuestro programa podría ser la siguiente:

```
vez = 1
while vez <= 5:
    Pasos1y2
    vez = vez + 1
```

Ahora bien, cuando finaliza la ejecución de la instrucción `vez = vez + 1` la estructura iterativa evalúa nuevamente la expresión `vez <= 5` cuyo resultado puede ser cierto o no (`True` o `False`). Si el resultado es `True`, entonces el ciclo continuará con las acciones contenidas, re-evaluando la expresión en cada iteración y finalizando cuando sea `False`, es decir, cuando la variable `vez` ya no sea menor o igual que 5.

Ahora que ya hemos desmenuzado el inofensivo código previo, podemos pasar a la versión final del pequeño programa.

```
vez = 1
while vez <= 5:
    temperatura = int(input('Ingrese la temperatura en °C:'))
    if (temperatura > 16):
        print('Vas caminando')
    else:
        print('Mucho frío, en vehículo')
    vez = vez + 1
```

```
Ingrese la temperatura en °C:12
Mucho frío, en vehículo
Ingrese la temperatura en °C:16
Mucho frío, en vehículo
Ingrese la temperatura en °C:17
Vas caminando
Ingrese la temperatura en °C:18
Vas caminando
Ingrese la temperatura en °C:20
Vas caminando
```

Este tipo de bucle, donde la cantidad de iteraciones depende de una condición es denominado como **bucles o lazos condicionales** y cuenta con dos características destacables:

- El valor a ser evaluado en la expresión debe estar definido

- En cada iteración el valor a ser evaluado en la expresión debe modificarse

El primer ítem evita obtener un mensaje de error, ya que no es posible evaluar una expresión con un valor que aún no ha sido definido, es decir, que no tiene asignado algún valor válido.

La segunda característica evita tener un **bucle infinito** y por ende un programa que nunca finalice. Este tipo de errores es más difícil de detectar, ya que a priori el ejemplo parecería correcto.

### Bucles interactivos

Otro tipo de bucle para el que la estructura *while* se adapta fácilmente es aquellos donde la repetición depende de un valor que ingresa el usuario, es decir, para aquellos programas donde la condición de corte o repetición sea interactiva. Veamos un ejemplo en el que se calcula el promedio a partir del ingreso por parte del usuario de valores numéricos enteros.

Pensemos una posible estrategia para su solución: el programa le solicitará ingresar un nuevo valor numérico mientras que el usuario ingrese *si*, a su vez deberá ir sumando estos valores y contándolos. Veamos el pseudocódigo del algoritmo mencionado:

```
Inicializar variable suma para sumar los números
Inicializar variable cant para contar los números
Inicializar variable mas_datos donde se almacenará la respuesta del usuario (si/no)
Mientras la variable mas_datos sea si:
    Leer en x el nuevo valor numérico
    Sumarlo a la variable suma
    Contarlo
    Preguntar al usuario si sigue ingresando números
Mostrar en pantalla el promedio
```

Ahora veamos lo directa que es la traducción del algoritmo al lenguaje Python:

```
suma = 0.0
cant = 0
mas_datos = 'si'
while mas_datos == 'si':
    x = int(input('Ingrese valor'))
    suma = suma + x
    cant = cant + 1
    mas_datos = input('¿Mas valores (si/no)?')
print('El promedio de valores es', suma/cant)
```

```
Ingrese valor12
¿Mas valores (si/no)?si
Ingrese valor3
¿Mas valores (si/no)?si
Ingrese valor44
¿Mas valores (si/no)?no
El promedio de valores es 19.666666666666668
```

La limitación que encontramos está dada por la incomodidad de tener que ingresar dos valores por ciclo, uno para el dato numérico y otro para controlar si el usuario desea continuar o no. En



ciertos casos puede ser la única alternativa, sin embargo, en otros se puede utilizar los bucles centinelas que se describen a continuación.

## Bucles centinelas

Otro tipo de bucles denominados centinelas, son aquellos donde la condición de corte tiene que ver con un valor que se diferencia del patrón que se ingresará y, será útil para discernir el momento en que corresponda continuar o bien finalizar la repetición.

Para el caso del cálculo del promedio, suponiendo que todos los valores serán siempre positivos podríamos tomar la estrategia de controlar que el valor ingresado sea mayor a cero para continuar la iteración. El pseudocódigo, sin detalles, sería similar al siguiente:

```
Leer en x el primer valor numérico
Mientras el valor x no sea el centinela:
    Sumarlo a la variable suma
    Contarlo
    Leer en x el nuevo valor numérico
Mostrar en pantalla el promedio
```

Veamos la implementación del amigable programa en Python:

```
suma = 0.0
cant = 0
x = int(input('Ingrese valor (negativo para salir)'))
while x > 0:
    suma = suma + x
    cant = cant + 1
    x = int(input('Ingrese valor (negativo para salir)'))
print('El promedio de valores es', suma/cant)
```

Se debe ser cuidadoso en mantener exactamente el mismo mensaje previo a ingresar al ciclo y en la última instrucción para dar al usuario una idea de continuidad viendo una y otra vez el mismo comportamiento.

Para el ejemplo expuesto, la limitación esta dada para aquellos casos donde se ingresen valores negativos para ser incluidos en el cálculo del promedio. Sin embargo, Python provee herramientas que permiten salvar este inconveniente.

El problema consiste en:

1. Solicitar al usuario ingrese el valor numérico o que presione *enter* para salir
2. Evaluar en la expresión de corte para iterar mientras que el valor ingresado no sea vacío
3. Realizar los cálculos

```
suma = 0.0
cant = 0
x = input('Ingrese valor (<enter> para salir)')
while x != '':
    suma = suma + eval(x)
    cant = cant + 1
```

```
x = input('Ingrese valor (<enter> para salir)')
print('El promedio de valores es', suma/cant)
```

```
Ingrese valor (<enter> para salir)12
Ingrese valor (<enter> para salir)-2
Ingrese valor (<enter> para salir)-3
Ingrese valor (<enter> para salir)23
Ingrese valor (<enter> para salir)2
Ingrese valor (<enter> para salir)
El promedio de valores es 6.4
```

El valor leído en *x* no se convierte en un número entero, sino que se lo mantiene como *str* hasta el momento de sumarlo a la variable *suma* utilizando la función *eval()*. Cuando el usuario presione enter el caracter en *x* será igual al caracter vacío y no ingresará al ciclo *while*.

### 4.2.2 Sentencia *for*

La sentencia *for* provee otro modo de realizar bucles repetitivos en Python. Si bien la elección de un bucle u otro muchas veces dependerá del gusto o preferencia del programador, para ciertos casos suele ser más cómoda una estructura que otra.

Veamos la sintaxis básica del bucle *for*:

```
for <var> in <secuencia>:
    accion1
    accion2
    ...
    accionN
```

El *for* ejecuta el bloque de acciones tantas veces como elementos contenga la *secuencia*, y en cada iteración la variable *var* almacenará uno a uno sus valores.

El significado de secuencia para Python puede variar desde cadenas de caracteres a listas de valores de tipos de datos ya vistos, simplificando la definición, podemos definir una secuencia como todo tipo o estructura de datos formada por elementos por los que se puede iterar.

Veamos un ejemplo, donde mostramos los caracteres de una cadena.

```
palabra = 'estimados'
for letra in palabra:
    print(letra)
```

```
e
s
t
i
m
a
d
o
s
```

Al analizar el ejemplo vemos que la variable *palabra* que contiene una **cadena de caracteres**, **funciona como una secuencia**, y la variable *letra* en cada iteración toma automáticamente el caracter subsiguiente.

### 4.2.3 Iteraciones sobre secuencias numéricas

Para iterar sobre secuencias numéricas combinamos el uso del `for` con la función `range()`. Veamos un ejemplo de una iteración sobre 3 valores:

```
for num in range(3):  
    print(num)
```

```
0  
1  
2
```

Cuando utilizamos la función `range()` con un único argumento como dato, para el ejemplo previo el número 3, nos genera una secuencia de 3 valores, comenzando desde 0 y avanzando de a un valor, es decir, con paso 1. Sin embargo, podemos cambiar este comportamiento indicando el valor inicial y final haciendo `range(inicio, fin)`, por ejemplo, se se desea iterar por valores numéricos entre 10 y 14:

```
for num in range(10,14):  
    print(num)
```

```
10  
11  
12  
13
```

Se debe notar que el valor final no es alcanzado en la iteración. También es posible indicarle el paso del incremento, como se deduce del ejemplo previo, al indicar solamente el valor inicial y final, se da por sentado que el incremento es de 1, cambiemos este comportamiento utilizando `range(inicio, fin, paso)`:

```
for num in range(10,19,2):  
    print(num)
```

```
10  
12  
14  
16  
18
```

Otra posibilidad es recorrer una secuencia numérica en sentido inverso, utilizando un incremento negativo y los valores de inicio y fin consistentes:

```
for num in range(19,10,-2):  
    print(num)
```

```
19
17
15
13
11
```

Del resultado previo queda en evidencia que se mantiene la coherencia respecto a excluir el último valor de la secuencia y a incluir el inicial.

Veamos un ejemplo que resolvimos anteriormente utilizando el `while`, ahora usando `for`:

```
for vez in range(5)
    temperatura = int(input('Ingrese la temperatura en °C:'))
    if (temperatura > 16):
        print('Vas caminando')
    else:
        print('Mucho frío, en vehículo')
```

Como vemos, nos preocupamos de la inicialización de la variable `vez` y de controlar su incremento, ya que esto se realiza automáticamente, por lo que para ciclos que conocemos de antemano la cantidad de iteraciones suele ser más simple y directo que el `while`.

La sentencia `for` en combinación con `range()` es una instrucción muy potente y flexible, más aún al ser combinadas con otro tipo de estructuras de datos como cadenas de caracteres y listas, que veremos en secciones posteriores.

## 4.3 Estructura de datos

### 4.3.1 Listas

Hasta aquí todo dato procesado, manipulado y operado ha sido almacenado en variables, sin embargo, para ciertos problemas no son suficientes. Supongamos un caso donde leemos una serie de temperaturas mensuales durante los últimos 10 años y que posteriormente queremos saber las temperaturas que han superado la media.

Si utilizamos variables, deberíamos leer los 120 valores para calcular el promedio y reingresar nuevamente las temperaturas mensuales para corroborar aquellas que superaron la media. Claramente el usuario de este programa no estará muy feliz de tener que tipear nuevamente la totalidad de los datos.

Para este tipo de problemas y muchos otros más existe una estructura más compleja y de gran utilidad denominada **lista**.

A diferencia de una variable que contiene un dato por vez, una lista puede almacenar varios en forma simultánea en diferentes posiciones, por lo que para referirnos a uno de ellos necesitamos especificarle el índice. Por ejemplo, en la siguiente lista denominada `tempC` hay almacenados tres valores numéricos flotantes, el primero está en la posición 0, el segundo en la posición 1 y, el tercero en la posición 2:

12.2	33.3	12.1
0	1	2

Para **declarar e inicializar** una lista vacía y otra con esos tres valores haremos:

```
# Lista vacia
vacía = []
# Lista con 3 valores flotantes
tempC = [12.2, 33.3, 12.1]
```

Para acceder a un elemento específico, debemos utilizar el identificador de la lista, seguido del índice entre corchetes (cualquier expresión entera), veamos un ejemplo donde realizamos las siguientes acciones:

1. Imprimir en pantalla el segundo valor (la posición 1 porque empezamos a contar desde 0)
2. Asignarle un nuevo valor que lo reemplace y volver a imprimirlo
3. Mostrar todo el contenido de la lista usando un bucle *for*
4. Mostrar aquellas temperaturas que superaron el promedio

```
# Elemento 1 de la lista
print("2do elemento:", tempC[1])

# Reemplaza el elemento 1 con 100
tempC[1] = 100
print("2do elemento modificado:", tempC[1])

# Lista completa y calculo de promedio
print("Lista:")
media = 0.0
for i in tempC:
    print(i)
    media = media + i
media = media/3

# Elementos que superan el promedio
for i in tempC:
    if i > media:
        print("La temperatura", i, "superó la media")
```

```
2do elemento: 100
2do elemento modificado: 100
Lista:
12.2
100
12.1
La temperatura 100 superó la media
```

Como se observa en el ciclo iterativo previo, las listas son perfectamente iterables en el `for`, ya que al igual que una cadena de caracteres, es una secuencia de valores, la diferencia radica

que en una cadena los valores son caracteres mientras que en una lista pueden ser de cualquier tipo y son llamados elementos o ítems.

Otro detalle es que una lista puede contener elementos de diferente tipo, incluso otra lista. Veamos una lista que combine elementos de distintos tipos:

```
# Lista que almacena distintos tipos de datos
popurri = [12, 3.1415, "amapola del 66", True, tempC]

# Imprimen los elementos
print("1er elemento: ", popurri[0])
print("2do elemento: ", popurri[1])
print("3er elemento: ", popurri[2])
print("4to elemento: ", popurri[3])
print("5to elemento: ", popurri[4])
```

```
1er elemento: 12
2do elemento: 3.1415
3er elemento: amapola del 66
4to elemento: True
5to elemento: [12.2, 100, 12.1]
```

Ahora bien, seguramente el lector estará intrigado sobre el acceso a un elemento en particular de la lista *tempC*, ubicada en la 5ta posición de la lista *popurri*. En *popurri[4]* se referencia el elemento en cuestión, que es una lista, por lo que agregando un índice más accedemos, veamos el código:

```
print(popurri[4][0])
print(popurri[4][1])
print(popurri[4][2])
```

```
12.2
100
12.1
```

Una de las funcionalidades que nos provee Python para obtener información sobre la cantidad de elementos de las listas es `len()`. Veamos los resultados que arroja aplicado a la lista *popurri*.

```
print(len(popurri))
print(len(popurri[4]))
```

```
5
3
```

Otra alternativa para iterar sobre una lista es combinando la función `range` que vimos anteriormente y la cantidad de elementos de la lista, de manera que podemos acceder a los ítems a partir de su índice:

```
n = len(tempC)
for i in range(n):
    print("Temperatura", i, ":", tempC[i])
```

```

Temperatura 0 : 12.2
Temperatura 1 : 33.3
Temperatura 2 : 12.1

```

La función `len()` retornó la cantidad de elementos de la lista *tempC*, ese resultado, almacenado en *n*, fué utilizado como el valor para la función `range()` que generó una secuencia numérica (una lista!!!) que va desde 0 hasta *n*-1.

Veamos otro ejemplo de una lista de cadenas de caracteres. Tenemos algunos equipos de fútbol santafesino de primera división y queremos imprimir el fixture con todas las combinaciones de los partidos de ida, es decir, si el equipo A ya jugó con el B, no tendremos en cuenta que el equipo B juegue con el A.

**Analicemos la estrategia:** Por cada equipo de la lista debemos imprimir uno a uno los rivales subsiguientes, es decir, imprimimos el primer equipo con el segundo, luego con el tercero y finalmente con el cuarto. Luego, al pasar al segundo equipo de la lista, no debemos imprimir el primero, porque ya fué rival, sino que los restantes y así sucesivamente.

```

equipos = ["Colón", "A. Rafaela", "Central", "Newell"]
n = len(equipos)
for i in range(n):
    for j in range(i+1,n):
        print(equipos[i], "vs", equipos[j])

```

```

Colón vs A. Rafaela
Colón vs Central
Colón vs Newell
A. Rafaela vs Central
A. Rafaela vs Newell
Central vs Newell

```

## Listas bidimensionales

Una lista unidimensional es aquella donde se utiliza un único índice para acceder a sus elementos, en el caso que utilizemos dos índices la lista es bidimensional y se la denomina matriz.

Veamos un caso de una lista bidimensional de tres filas y cinco columnas (3x5)

	0	1	2	3	4
0	12.2	33.3	12.1	0.3	1.21
1	3.14	2.1	9.8	28.1	19.8
2	10.8	0.1	0.2	22.1	9.38

Veamos el modo de definirla:

```

matriz = [
    [12.2, 33.3, 12.1, 0.3, 1.21],
    [3.14, 2.1, 9.8, 28.1, 19.9],
    [10.8, 0.1, 0.2, 22.1, 9.38]
]

```

El acceso a cada dato se realiza utilizando los dos índices, donde el primero hace referencia a la fila y el segundo a la columna. Así, si se accede al segundo elemento (1) de la tercer fila sería (2): `matriz[2][1]`.

El recorrido de una matriz se simplifica utilizando ciclos repetitivos anidados, veamos un posible modo de iterar por las columnas de la matriz previamente definida.

```
for c in range(5):
    print("Columna", c)
    for f in range(3):
        print(matriz[f][c])
    print()
```

```
('Columna', 0)
12.2
3.14
10.8
()
('Columna', 1)
33.3
2.1
0.1
()
('Columna', 2)
12.1
9.8
0.2
()
('Columna', 3)
0.3
28.1
22.1
()
('Columna', 4)
1.21
19.9
9.38
()
```

## Operaciones

En Python, las listas, las tuplas y las cadenas de caracteres son parte del conjunto de las secuencias. Todas las secuencias cuentan con las siguientes operaciones:



Operación	Resultado
<code>x in s</code>	Indica si la variable <code>x</code> se encuentra en <code>s</code>
<code>s + t</code>	Concatena las secuencias <code>s</code> y <code>t</code> .
<code>s * n</code>	Concatena <code>n</code> copias de <code>s</code> .
<code>s[i]</code>	Elemento <code>i</code> de <code>s</code> , empezando por 0.
<code>s[i:j]</code>	Porción de la secuencia <code>s</code> desde <code>i</code> hasta <code>j</code> (no inclusive).
<code>s[i:j:k]</code>	Porción de la secuencia <code>s</code> desde <code>i</code> hasta <code>j</code> (no inclusive), con paso <code>k</code> .
<code>len(s)</code>	Cantidad de elementos de la secuencia <code>s</code> .
<code>min(s)</code>	Mínimo elemento de la secuencia <code>s</code> .
<code>max(s)</code>	Máximo elemento de la secuencia <code>s</code> .

## Rebanadas (slices)

Para acceder a los elementos de una lista se puede usar como índice cualquier expresión entera, por lo que `tempC[1+1]` o `matriz[2*0+1][2*2]` son operaciones perfectamente válidas. Además, se pueden extraer conjuntos de elementos de la lista a partir de porciones o rebanadas (slices). Veamos unos ejemplos.

```
>>> lista = ['a', 'b', 'c', 'd', 'e', 'f']
>>> lista[1:3]
['b', 'c']
>>> lista[:4]
['a', 'b', 'c', 'd']
>>> lista[3:]
['d', 'e', 'f']
>>> lista[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Podemos reemplazar varios elementos a la vez:

```
>>> lista = ['a', 'b', 'c', 'd', 'e', 'f']
>>> lista[1:3] = ['x', 'y']
>>> print lista
['a', 'x', 'y', 'd', 'e', 'f']
```

Además, puede eliminar elementos de una lista asignándoles la lista vacía:

```
>>> lista = ['a', 'b', 'c', 'd', 'e', 'f']
>>> lista[1:3] = []
>>> lista
['a', 'd', 'e', 'f']
```

Y se puede añadir elementos a la lista insertándolos en una porción vacía en la posición deseada:

```
>>> lista = ['a', 'd', 'f']
>>> lista[1:1] = ['b', 'c']
>>> print lista
['a', 'b', 'c', 'd', 'f']
>>> lista[4:4] = ['e']
>>> print lista
['a', 'b', 'c', 'd', 'e', 'f']
```

### Métodos

Una lista provee una serie de funcionalidades asociadas denominados métodos. Se propone profundizar sobre los métodos disponibles con la lectura del *Tutorial de Python* (pág. 26, *Más sobre listas*)

- `list.append(x)` Agrega un ítem al final de la lista. Equivale a `a[len(a):] = [x]`
- `list.extend(L)` Extiende la lista agregándole todos los ítems de la lista dada. Equivale a `a[len(a):] = L`
- `list.insert(i, x)` Inserta un ítem en una posición dada. El primer argumento es el índice del ítem delante del cual se insertará, por lo tanto `a.insert(0, x)` inserta al principio de la lista, y `a.insert(len(a), x)` equivale a `a.append(x)`
- `list.remove(x)` Quita el primer ítem de la lista cuyo valor sea `x`. Es un error si no existe tal ítem
- `list.pop([, i])` Quita el ítem en la posición dada de la lista, y lo devuelve. Si no se especifica un índice `a.pop()` quita y devuelve el último ítem de la lista. (Los corchetes que encierran a `i` en la firma del método denotan que el parámetro es opcional, no que deberías escribir corchetes en esa posición. Verás esta notación con frecuencia en la Referencia de la Biblioteca de Python.)
- `list.clear()` Quita todos los elementos de la lista. Equivalente a `del a[:]`
- `list.index(x)` Devuelve el índice en la lista del primer ítem cuyo valor sea `x`. Es un error si no existe tal ítem
- `list.count(x)` Devuelve el número de veces que `x` aparece en la lista
- `list.sort()` Ordena los ítems de la lista in situ
- `list.reverse()` Invierte los elementos de la lista in situ
- `list.copy()` Devuelve una copia superficial de la lista. Equivalente a `a[:]`

Una manera de quitar un ítem de una lista dado su índice en lugar de su valor es la instrucción `del`, que también puede usarse para quitar secciones de una lista o vaciar la lista completa. Por ejemplo:

```
a = [-1, 1, 66.25, 333, 333, 1234.5]
del a[0]
a
[1, 66.25, 333, 333, 1234.5]
del a[2:4]
a
[1, 66.25, 1234.5]
```

### 4.3.2 Diccionarios

Hemos visto que las listas son útiles cuando se quiere agrupar valores en una estructura y acceder a cada uno de ellos a través de un valor numérico, un índice.

Otro tipo de estructura, que nos permite referirnos a un determinado valor a través de un nombre es un diccionario. Muchas veces este tipo de estructura es más apropiado que una lista.

El nombre *diccionario* da una idea sobre el propósito de esta estructura ya que uno puede realizar fácilmente una búsqueda a partir de una palabra específica (*clave*) para obtener su definición (*valor*).

Un ejemplo podría ser una agenda telefónica, que nos permita obtener el número de teléfono de una persona a partir de su nombre. Veamos entonces el modo de crear diccionarios.

```
agenda = {'Marado':'1552123', 'JPFeinman':'1523443', 'Dolina':'4584129',
          'Spasiuk':'65748', 'Fontanarrosa':'32456'}
```

Los *diccionarios* consisten en pares (llamados *items*) de *claves* y sus *valores* correspondientes. En este ejemplo, los nombres son las claves y los números de teléfono son los valores. Cada clave es separada de su valor por los puntos (:), los items son separados por comas, y toda la estructura es encerrada entre llaves. Un diccionario vacío, sin items, se escribe con solo dos llaves: {}.

Las claves, debido a que funcionan como índices, no pueden ser repetidas.

Veamos las formas más comunes de iterar sobre un diccionario:

```
# Imprime claves
print("Claves")
print("=====")
for nom in agenda:
    print(nom)
print()

print("Valores")
print("=====")
# Imprime valores
for tel in agenda.values():
    print(tel)
print()

print("Clave y valor")
print("=====")
# Imprime items: clave valor
for nom, tel in agenda.items():
    print(nom, tel)
```

```
Claves
=====
JPFeinman
Spasiuk
Marado
Dolina
Fontanarrosa

Valores
=====
```

```
1523443
65748
1552123
4584129
32456

Clave y valor
=====
JPFeinman 1523443
Spasiuk 65748
Marado 1552123
Dolina 4584129
Fontanarrosa 32456
```

Al igual que las listas, los diccionarios son sumamente flexibles y pueden estar formados por otros diccionarios (o inclusive listas). Analicemos un breve ejemplo de un diccionario que está conformado del siguiente modo:

- Cuenta con tres items
- El valor de cada item es otro diccionario que a su vez contiene:
  - Tres items con las claves *título*, *fecha* y *autor*

A continuación veamos la implementación de esta estructura, la impresión manual y mediante iteración:

```
referencia = { "libro1":{"titulo":"El tutorial de Python",
                        "fecha":"2013",
                        "autor":"Guido van Rossum"},
               "libro2":{"titulo":"Aprenda a Pensar Como un Programador con Python",
                        "fecha":"2002",
                        "autor":"Allen Downey"},
               "libro3":{"titulo":"Inmersión en Python 3",
                        "fecha":"2009",
                        "autor":"Mark Pilgrim"}
             }

# acceso a los valores de titulo de cada libro
print("Títulos")
print("=====")
print(referencia["libro1"]["titulo"])
print(referencia["libro2"]["titulo"])
print(referencia["libro3"]["titulo"])
print()

# Mezcladito
for clave in referencia:
    print(clave)
    print("=====")
    for clave2, val in referencia[clave].items():
        print(clave2, val, sep=": ")
    print()
```

```
Titulos
=====
El tutorial de Python
Aprenda a Pensar Como un Programador con Python
Inmersión en Python 3

libro3
=====
autor: Mark Pilgrim
titulo: Inmersión en Python 3
fecha: 2009

libro2
=====
autor: Allen Downey
titulo: Aprenda a Pensar Como un Programador con Python
fecha: 2002

libro1
=====
autor: Guido van Rossum
titulo: El tutorial de Python
fecha: 2013
```

## Operaciones

- `len(d)` retorna el número de items (pares clave-valor) en `d`
- `d[k]` retorna el valor asociado con la clave `k`
- `d[k] = v` asocia el valor `v` con la clave `k`
- `del d[k]` elimina el item con clave `k`
- `k in d` evalúa si existe un item en `d` que tenga la clave `k`

Aunque las listas y los diccionarios comparten varias características en común, existen ciertas distinciones importantes:

- Tipos de claves: Las claves de los diccionarios no deben ser enteros (aunque pueden serlo). Deben ser tipos de datos inmutables (números flotantes, cadenas de caracteres o tuplas)
- Agregado automático: En un diccionario se crea un item automáticamente al asignar un valor a una clave inexistente, en una lista no se puede agregar un valor en un índice que esté fuera del rango.
- Contenido: La expresión `k in d` (`d` es un diccionario) evalúa por la existencia de una clave, no de un valor. Por otro lado, la expresión `v in l` (siendo `l` una lista), busca por un valor en vez de por un índice.

### Métodos

A continuación se describen brevemente algunos de los métodos más utilizados:

`clear()`

Elimina todos los items

`copy()`

Retorna una copia superficial del diccionario

`get(key[, default])`

Retorna el valor de la clave `key` si existe, sino el valor `default`. Si no se prop

`items()`

Retorna el par de valores del item `clave, valor`.

`keys()`

Retorna las claves.

`pop(key[, default])`

Si la clave `key` está presente en el diccionario la elimina y retorna su valor, s

`popitem()`

Elimina y retorna un par (`clave, valor`) arbitrario.

`setdefault(key[, default])`

Si la clave `key` está presente en el diccionario retorna su valor. Si no, inserta

`update([other])`

Actualiza los items de un diccionario en otro. Es útil para concatenar diccionar

`values()`

Retorna los valores del diccionario.

Los diccionarios pueden ser comparados por su igualdad si y solo si tienen los mismos items. Otras comparaciones (`'<'`, `'<='`, `'>='`, `'>'`) no son permitidas.

Para profundizar sobre diccionarios se recomienda la lectura del *Tutorial de Python* (pág. 32, *Diccionarios*).

### 4.3.3 Tuplas

Las tuplas son secuencias, al igual que las listas. La única diferencia es que no pueden ser modificadas, son inmutables (al igual que las cadenas de caracteres).

La sintaxis de las tuplas es simple, al separar varios valores con comas, automáticamente se crea una tupla.

```
t = 28, 21, 'hola!'
print(t[0])
print(t)

# desempaquetado de una tupla
x, y, z = t
```

```
28
(28, 21, 'hola!')
```

Para mayor detalle sobre esta estructura se recomienda leer el Tutorial de Python, *Tuplas y secuencias*, pag. 30.

### 4.3.4 Conversión entre listas y diccionarios

#### De diccionarios a listas

Es posible crear listas a partir de diccionarios usando los métodos `items()`, `keys()` y `values()`. El método `keys()` crea una lista que consiste solamente en las claves del diccionario, mientras que `values()` produce una lista que contiene los valores. `items()` puede ser usado para crear una lista que conste de tuplas de dos pares (clave, valor). Utilicemos el diccionario agenda creado anteriormente:

```
print("Lista de items")
print("=====")
items_vista = agenda.items()
items = list(items_vista)
print(items)
print()

print("Lista de claves")
print("=====")
claves_vista = agenda.keys()
nombres = list(claves_vista)
print(nombres)
print()

print("Lista de valores")
print("=====")
valores_vista = agenda.values()
telefonos = list(valores_vista)
print(telefonos)
```

```
Lista de items
=====
[('Dolina', '4584129'), ('Fontanarroza', '32456'), ('JPFeinman', '1523443'), ('S
```

```
Lista de claves
=====
['Dolina', 'Fontanarrosa', 'JPFeinman', 'Spasiuk', 'Marado']

Lista de valores
=====
['4584129', '32456', '1523443', '65748', '1552123']
```

### De listas a diccionarios

Ahora realizaremos el proceso inverso, para armar un diccionario a partir de dos listas. Ya en el ejemplo previo obtuvimos dos listas, una con los nombres y otra con los teléfonos. Las funciones a utilizar son 3: `zip()`, `list()` y `dict()`. Veamos:

```
lista_de_tuplas = list(zip(nombres, telefonos))
agenda2 = dict(lista_de_tuplas)
print(agenda2)
```

```
{'JPFeinman': '1523443', 'Fontanarrosa': '32456', 'Dolina': '4584129', 'Spasiuk': '1552123'}
```

### 4.3.5 Cadenas de caracteres

Una cadena es una secuencia de caracteres. Las hemos usado para mostrar mensajes, pero sus usos son mucho más amplios que sólo ése, a continuación las veremos mas en profundidad.

Es importante destacar:

- Las cadenas son inmutables: una vez creadas no podemos modificarlas accediendo manualmente a sus caracteres.
- El acceso a sus caracteres es igual al de los elementos de una lista. El primer caracter se encuentra en la posición cero y soporta el indexado y las rebanadas o porciones tal como las listas.

Veamos la siguiente cadena:

```
frase = 'siento que nací en el viento'
```

- Obtenemos la cantidad de caracteres utilizando la función `len(frase)`
- Accedemos a los caracteres usando índices, por ejemplo, el cuarto caracter se encuentra en `frase[3]`
- Soporta rebanadas, podemos extraer por ejemplo la segunda palabra, `frase[7:10]`
- La última palabra: `frase[-6:]`



## Operaciones

Hemos visto ya dos operadores matemáticos que son compatibles para su uso con cadenas de caracteres: operador suma (+) y el multiplicación (\*). Recordemos su funcionamiento con un simple ejemplo

```
w = "libertad"
print(3*(w+' '))
```

```
libertad libertad libertad
```

Las cadenas de caracteres pueden ser comparadas entre si mediante los símbolos: >, >=, <, <=, ==, !=. Veamos un ejemplo:

```
palabra = input("Ingresa una palabra: ")
if palabra < w:
    print("Tu palabra, "+palabra+ " , va antes que " + w)
elif palabra > w:
    print("Tu palabra, "+palabra+ " , va después que " + w)
else:
    print("Tu palabra, "+palabra+ " , es " + w)
```

```
Ingresa una palabra: cadenas
Tu palabra, cadenas, va antes que libertad
```

## Métodos

Las cadenas también cuentan con métodos que realizan una función específica, a continuación vemos los más usuales:

find

```
Busca una subcadena dentro de otra.
```

lower y upper

```
Retorna la cadena en minúsculas
```

replace

```
Retorna una cadena donde todas las ocurrencias de una cadena son reemplazadas por
```

split

```
Separa una cadena según un caracter separador y retorna una lista con los elementos
```

strip

```
Retorna una cadena donde los espacios en blanco al inicio y al final de la cadena
```

join

Es el inverso de `split`. Une elementos de una lista en una cadena de caracteres u

Apliquemos algunos de estos métodos:

```
print(frase.find("nací"))
print(frase.lower())
print(frase.upper())
print(frase.replace("viento", "hospital"))
lista_frase = frase.split(" ")
print(lista_frase)
sep = "-"
print(sep.join(lista_frase))
```

```
11
siento que nací en el viento
SIENTO QUE NACÍ EN EL VIENTO
siento que nací en el hospital
['siento', 'que', 'nací', 'en', 'el', 'viento']
siento-que-nací-en-el-viento
```

---

## Ejercicios

---

1. Utilice una estructura repetitiva for para iterar sobre las letras de una palabra y muestre en pantalla su versión encriptada. Para encriptarla imprima en pantalla el reemplazo de una letra con un número según lo siguiente: a->4, b->8, e->3, f->7, t->2, g->9, i->1, o->0.
2. Se lee una cadena de caracteres por teclado y se pide que la traduzca a código morse utilizando el siguiente diccionario como base:

```
morse = {  
"A" : ".-",  
"B" : "-...",  
"C" : "-.-.",  
"D" : "-..",  
"E" : ".",  
"F" : ".-.-",  
"G" : "--.",  
"H" : "....",  
"I" : "..",  
"J" : ".---",  
"K" : "-.-",  
"L" : ".-..",  
"M" : "--",  
"N" : "-.",  
"O" : "---",  
"P" : ".-.-",  
"Q" : "--.-",  
"R" : "-.-",  
"S" : "...",  
"T" : "-",  
"U" : ".-.-",  
"V" : "...-",  
"W" : ".--",  
"X" : "-.-.-",  
"Y" : "-.-.-",  
"Z" : "--..",  
"0" : "-----",  
"1" : ".-----",  
"2" : "..-----",
```

```
"3" : "...--",
"4" : "...-",
"5" : ".....",
"6" : "-.....",
"7" : "--....",
"8" : "---...",
"9" : "----.",
"." : ".-.-.-",
", " : "--.---"
}
```

### Tabla de Contenidos

```
%%javascript
$.getScript('https://kmahelona.github.io/ipython_notebook_goodies/ipython_notebo
```

```
<IPython.core.display.Javascript object>
```

---

## Funciones y archivos

---

### 6.1 Funciones

Las funciones son subprogramas que pueden ser invocados para realizar una tarea específica, siendo capaz de recibir información (datos de entrada) desde donde son llamados y a su vez de retornar algún valor (datos de salida).

Cuando una función es invocada, el programa principal transfiere el control a la función hasta que finalice su ejecución, volviendo luego al punto desde donde fue llamada.

En los programas desarrollados anteriormente hicimos uso de la función `len()`, que recibe como información de entrada una secuencia (lista o una cadena de caracteres por ejemplo) y retorna un valor numérico entero que representa la cantidad de elementos (o de caracteres). `len()` es una de las tantas funciones prefdefinidas por el lenguaje Python en la biblioteca estándar y, en la presente sección veremos cómo definir nuestras propias funciones.

El uso de funciones en el desarrollo de programas tiene un conjunto de ventajas, dentro de las que se destacan:

- Subdividir un problema complejo en problemas mas simples: divide y vencerás.
- Mejoran la legibilidad del código, los programas modulares son más fáciles de mantener y entender.
- Posibilitan la reusabilidad del código, llamar funciones desde distintos programas.

#### 6.1.1 Definición y uso

Una función se define anteponiendo la palabra clave `def` seguida del nombre de la función, paréntesis de apertura y cierra y los dos puntos (:). Luego, el bloque de acciones que la conforman. Veamos la estructura:

```
def nombre_funcion(argumento1, argumento2, ..., argumentoN):  
    accion1  
    accion2  
    ...  
    accionN
```

En el caso previo la función recibe como entrada argumentos y realiza una serie de acciones. Las funciones pueden ser definidas en el mismo programa, con la finalidad de organizar mejor el código. Veamos un ejemplo de un programa que define y utiliza una función denominada `muestra_doble()`.

```
# Definición de la función
def muestra_doble(x):
    '''Imprime en pantalla el doble de x'''
    print(2*x)

# Programa principal
a = 3.5
# invoca a la función
muestra_doble(a)
print('Todo OK')
```

```
7.0
Todo OK
```

Analicemos en detalle la secuencia de ejecución:

- Desde el programa principal se invoca a la función enviando la variable `a` como parámetro
- La función recibe la entrada haciendo una copia de `a` en la variable `x`
- La función ejecuta sus acciones y vuelve el control al programa principal
- El programa principal continúa la ejecución hasta finalizar

Como vemos, la función no ha retornado valor alguno al programa principal, modifiquemos la función de manera que en vez de imprimir en pantalla el doble del valor, lo retorne al programa principal.

```
# Definición de la función
def calc_doble(x):
    """Retorna el doble de x"""
    return 2*x

# Programa principal
a = 3.5
# invoca a la función
doble = calc_doble(a)
print(doble)
```

```
7.0
```

Al igual que en el ejemplo anterior, la función es invocada desde el programa principal con el parámetro `a` y es copiado automáticamente como `x` dentro de la función. Destaquemos las diferencias:

- El programa principal invoca la función desde una asignación (`doble = calc_doble(a)`)
- Antes de realizarse la asignación, la ejecución pasa el control a la función.

- La función realiza las acciones programadas y al ejecutar la palabra reservada `return` asigna la operación a su nombre y vuelve el control al programa principal
- El nombre de la función contiene el resultado y es asignado a la variable `doble`
- Finaliza el programa

Si bien parece trivial, es importante que el nombre de la función sea acorde a las acciones que realiza e identifique su comportamiento, por este motivo la función fue renombrada a `calc_doble`.

El retorno de valores de una función es completamente flexible, se pueden retornar más de una variable, listas, tuplas, diccionarios o cualquier combinación de ellas. Veamos un caso de una función que recibe dos listas de nombres y teléfonos y retorna una agenda en una estructura de diccionario, donde la primera lista conforma las claves y la segunda los valores.

```
def arma_agenda(lista_nom, lista_tel):
    '''recibe 2 listas y retorna un diccionario'''
    d = {}
    for nom, tel in zip(lista_nom, lista_tel):
        d[nom] = tel
    return d

# Programa principal
n = ['Kliksberg', 'Stiglitz', 'Zaffaroni']
t = ['23444', '54556', '66554']
agenda = arma_agenda(n,t)
print(agenda)
```

```
{'Kliksberg': '23444', 'Stiglitz': '54556', 'Zaffaroni': '66554'}
```

El lector atento habrá notado que en todas las funciones debajo de su definición existe un texto encerrado entre comillas triples (como por ejemplo `"""Retorna el doble de x"""`). Esto es un comentario que se utiliza para documentar brevemente, y con nuestras palabras, que es lo que realiza dicha función. Su uso es opcional, pero es muy recomendable, dado que puede ser de mucha utilidad tanto para nosotros como para otros desarrolladores.

## 6.1.2 Variables globales y locales

Hemos visto que las funciones reciben un conjunto de valores a través de sus parámetros, sin embargo no fueron modificados dentro de la función. La pregunta que surge es: ¿Podemos cambiarlos? ¿Qué sucede si los modificamos?

Veamos un ejemplo y su comportamiento:

```
def trata_de_cambiar(nombre):
    nombre = 'Luis Alberto Spinetta'

n = 'Norberto Napolitano'
trata_de_cambiar(n)
print(n)
```

Norberto Napolitano

Observamos que la variable no fué modificada o al menos no se ve reflejado desde el programa principal. Esto sucede debido a que la variable `n` es copiada en la variable `nombre` y todo cambio que se realice en el interior de `trata_de_cambiar` será local, es decir, su ámbito de validez se limita a la función, de manera tal que tanto `Spinetta` como `Napolitano` son irremplazables.

No obstante, existen estructuras de datos que al ser modificadas dentro la función su cambio se verá reflejado en el programa principal. La única condición para que sea posible este comportamiento es que la estructura a ser modificada como argumentos sea *mutable*, tal es el caso de los diccionarios y listas.

Veamos un caso donde definimos una función que recibe dos argumentos, una cadena de caracteres y una lista, de tipo *immutable* y *mutable* respectivamente.

```
def todo_cambia(musico, listam):
    listam.append(musico)

artistas = []

todo_cambia('Luis Alberto Spinetta', artistas)
todo_cambia('Chango Spasiuk', artistas)
todo_cambia('Norberto Napolitano', artistas)
todo_cambia('Charly García', artistas)

print(artistas)
```

```
['Luis Alberto Spinetta', 'Chango Spasiuk', 'Norberto Napolitano', 'Charly García']
```

El primer argumento, `musico`, es una cadena de caracteres que contiene el nombre de un artista y el segundo argumento, `listam`, es una lista donde se agrega el músico.

Es importante notar que el ejemplo es equivalente al anterior, la diferencia radica únicamente en que el argumento que es modificado en la función es la misma lista del programa principal, no una copia, independientemente que en el programa principal utilice un identificador diferente al de la función.

Ahora bien, existen casos donde es necesario modificar una variable del programa principal desde una función sin que sea recibida a través de sus argumentos. Para realizar este tipo de acciones necesitamos utilizar variables cuyo ámbito de validez sea tanto el programa principal como la función, es decir, variables globales.

Veamos un ejemplo de una función que incrementa una variable global cuando el número que recibe por argumentos es par:

```
def contar(num):
    global pares
    if num % 2 == 0:
        pares = pares + 1

pares = 0
```



```

contar(2)
contar(5)
contar(8)

print(pares)

```

2

Algunos detalles a destacar sobre variables globales:

- Se debe anteponer a la variable la palabra reservada `global`
- Toda modificación repercutirá en el programa principal

El uso de variables globales es una práctica que generalmente debe ser evitada. En la mayoría de los casos es preferible utilizar un parámetro y que la función retorne en su nombre el valor modificado.

### 6.1.3 Agrupando el código en módulos

Hemos visto como organizar mejor el código a través de funciones, sin embargo, una de las ventajas de utilizar funciones propias es evitar la reescritura. Carece de sentido tener que reprogramar una misma función por cada programa y, por otro lado, con el paso del tiempo es muy probable que no todas las versiones sean idénticas y por ende, su comportamiento puede diferir.

Para solucionar este tipo de problemas y sacar provecho del uso de funciones existen los módulos, cuya utilidad es la de contener varias funciones que realicen algún tipo de tarea afin.

Por ejemplo, una serie de funciones para cálculo matemático sería útil que estén contenidas en un mismo módulo, otras funciones para procesamiento de sonido en un módulo destinado a tal fin, o bien, una serie de funciones destinadas a almacenar todas las funciones relativas a un determinado proyecto.

Para comprender la implementación veamos un módulo trivial, que contenga saludos en diferentes idiomas. Almacenamos en el archivo `saludo.py` las siguientes funciones:

```

def espanol(nom):
    print('Hola', nom)

def quechua(nom):
    print('Napaykullayki', nom)

def italiano(nom):
    print('Ciao', nom)

def guarani(nom):
    '''Buen dia, cómo estas?'''
    print("Mba'éichapa ndepyhareve", nom)

def aymara(nom):

```

```
'''¿cómo estás?'''  
print('Kamisaraki', nom)  
  
def maya(nom):  
    '''¿cómo estás?'''  
    print('Biix yanilech?', nom)
```

Luego, creamos el programa desde donde será importado el módulo e invocadas las funciones que contiene. Por ejemplo, en `charlando.py` hacemos lo siguiente:

```
import saludo  
  
n = input('Ingrese su nombre: ')  
saludo.italiano(n)  
saludo.guarani(n)
```

Como observamos, el módulo es importado a través del nombre del archivo (sin la extensión `.py`) y luego, se invocan las funciones utilizando el nombre del módulo y la función separado por un punto (`.`).

De esta manera, tenemos acceso a la totalidad de las funciones definidas bajo el módulo, pero, para el caso que únicamente se utilice una función específica, es posible especificarlo en la cláusula `import` del siguiente modo:

```
from saludo import italiano, guarani  
  
n = input('Ingrese su nombre: ')  
italiano(n)  
guarani(n)
```

De esta manera, es posible invocar solamente las funciones importadas.

## 6.2 La biblioteca estándar

Se recomienda la lectura del capítulo *Pequeño paseo por la Biblioteca Estándar. Parte I* (pag. 72) del Tutorial de Python.

## 6.3 Archivos

Hasta aquí hemos trabajado con información almacenada en estructuras de datos, ya sea a partir de la lectura interactiva (utilizando la función `input`) o cargada estáticamente en el mismo código del programa y, la salida ha sido siempre a través de la impresión en pantalla (utilizando la función `print`).

La limitación de este modo de trabajo es que la información no se almacena de modo persistente. Para resolver este inconveniente veremos en la presente sección la manera de utilizar información de entrada y salida para nuestros programas a través de archivos de texto.

Incorporar el uso de archivos a un programa generalmente requiere las siguientes acciones:

- Abrir el archivo: la apertura de un archivo se realiza a partir de la primitiva `open` y consiste en asociar un elemento del programa con un archivo en particular.
- Elegir el modo de apertura: un archivo puede abrirse para lectura (r), escritura (w), agregado (a), binario (b), lectura/escritura (+)
- Leer ó escribir en el archivo
- Cerrar el archivo

Trabajemos con un archivo de texto, por ejemplo `archi01.txt`, con el siguiente contenido:

```
enero 30
febrero 60
marzo 55
```

### 6.3.1 Lectura

Vamos a realizar la lectura de este archivo e imprimir por pantalla su contenido. Dos de los métodos más comunes son:

- `readline()`: lee de a una línea por vez
- `readlines()`: lee todo el contenido del archivo y lo retorna en una lista

Veamos como sería el funcionamiento del primer caso:

```
# Apertura del archivo en modo lectura
f = open('ejemplos/u4/archi01.txt', 'r')

# Lee la primer línea
r = f.readline()
print(r)

# Lee la segunda línea
r = f.readline()
print(r)

# Cierra el archivo
f.close()
```

```
enero 30

febrero 60
```

Probablemente sea más práctico realizar la lectura línea por línea en un ciclo iterativo hasta que se llegue al final del archivo. Esto se puede realizar combinando lo anterior con un ciclo repetitivo `while`:

```
# Apertura del archivo en modo lectura
f = open('ejemplos/u4/archi01.txt', 'r')
```

```
# Lee la primer línea
r = f.readline()
while r:
    print(r)
    # lee la sgte
    r = f.readline()
f.close()
```

```
enero 30

febrero 60

marzo 55
```

En este caso, la función `readline` retornara `False` cuando se llegue al final del archivo, y por lo tanto se saldrá del ciclo `while`. Otro método más directo y elegante -en general preferido- para realizar un comportamiento equivalente (agregado desde la versión de Python 2.2) es iterar sobre los mismos archivos, esto es:

```
# Apertura en modo lectura (por defecto)
f = open('ejemplos/u4/archi01.txt')

for r in f:
    print(r)
f.close()
```

```
-----
IOError                                Traceback (most recent call last)

<ipython-input-2-048e5e9434f7> in <module>()
      1 # Apertura en modo lectura (por defecto)
----> 2 f = open('ejemplos/u4/archi01.txt')
      3
      4 for r in f:
      5     print(r)

IOError: [Errno 2] No such file or directory: 'ejemplos/u4/archi01.txt'
```

El método `readlines()` lee el contenido completo del archivo retornando una lista con su contenido, donde cada elemento corresponde a un renglón del archivo.

Este método es más directo y suele ser útil para archivos que no son excesivamente grandes. Veamos un ejemplo:

```
# Apertura del archivo en modo lectura
f = open('ejemplos/u4/archi01.txt', 'r')

# Lee todo el archivo
todo = f.readlines()
```

```
# 1er linea
print(todo[0])

# lista con todo el contenido
print(todo)

f.close()
```

```
enero 30
```

```
['enero 30n', `febrero 60n`, `marzo 55n']
```

Ahora bien, podemos procesar los datos que son leídos del archivo. Hagamos el cálculo de un promedio con los valores numéricos de cada mes, para esto debemos extraer de la cadena de caracteres solamente aquellos valores que siguen a la cadena de caracteres correspondiente al mes. Para esto haremos uso de la función `split()`:

```
# Apertura del archivo en modo lectura
f = open('ejemplos/u4/archi01.txt', 'r')

# Lee todo el archivo
todo = f.readlines()

# para promedio
acum = 0
cont = 0

for r in todo:
    mes, val = r.split()    # separo por espacio
    acum = acum + int(val)  # sumo convirtiendo a entero
    cont = cont + 1        # cuento los valores

f.close()
promedio = acum/cont
print('Promedio: ', promedio)
```

```
Promedio:  48.333333333333336
```

### 6.3.2 Escritura

Para escribir datos en un archivo, inicialmente se lo abre para escritura, luego se pueden utilizar dos métodos:

- `write(r)`: escribe el contenido de `r` en un renglón del archivo
- `writelines(L)`: escribe el contenido completo de la lista `L` en el archivo

Veamos un ejemplo de `write`:

```
# Crea archivo en modo escritura
f = open('ejemplos/u4/archi02.txt', 'w')
```

```
# Lee todo el archivo
r1 = 'nace una flor\n'
f.write(r1)
r1 = 'todos los dias\n'
f.write(r1)
r1 = 'sale el sol\n'
f.write(r1)

f.close()
```

El programa creó el archivo y luego escribió los tres renglones. Se debe notar que al final de cada cadena se utilizó el caracter especial `\n` que se traduce en un salto de línea, sino cada texto se hubiese escrito a continuación.

Ahora veremos un ejemplo haciendo uso del método `writelines()`:

```
# Crea archivo en modo escritura
f = open('ejemplos/u4/archi03.txt', 'w')

# Lee todo el archivo
L = ['nace una flor\n', 'todos los dias\n', 'sale el sol\n']
f.writelines(L)

f.close()
```

Como se observa, al igual que en el método anterior se debe agregar el caracter especial de retorno de línea al finalizar cada cadena. Se debe tener en cuenta que de no existir el archivo es creado pero, es borrado su contenido en caso contrario, por lo que debe prestarte especial atención para evitar la pérdida de datos involuntaria.

En aquellos casos donde sea necesario agregar contenido a un archivo ya existente entonces se debe utilizar el modo de apertura `a` (proveniente de Append). Veamos un ejemplo en el que se agregan unas líneas de datos al archivo `archi01.txt`.

```
# Abre archivo en modo append
f = open('ejemplos/u4/archi01.txt', 'a')

# Lee todo el archivo
L = ['abril 33\n', 'mayo 21\n', 'junio 88\n']
f.writelines(L)

f.close()
```

Finalmente el archivo quedará con el siguiente contenido:

```
enero 30
febrero 60
marzo 55
abril 33
mayo 21
junio 88
```

Es muy importante recordar que siempre debemos cerrar el archivo una vez que hemos trabajado con el mismo (función `close()`), independientemente de si lo hemos utilizado para lectura o para escritura.





---

## Ejercicios

---

1 . Escriba una función que reciba como parámetro la temperatura expresada en grados centígrados (°C) e imprima en pantalla la misma temperatura en grados Fahrenheit (°F). Tenga en cuenta que la formula de conversión de Celsius a Fahrenheit es la siguiente:

$$F = (9C/5,0) + 32$$

2 . Escriba un programa que lea un archivo con temperaturas expresadas en grados centígrados, y que luego genere otro archivo que contenga dichas temperaturas pero expresadas en grado Fahrenheit. Como archivo de entrada, utilice el siguiente:

```
10 °C
12.5 °C
14.2 °C
16.6 °C
18.2 °C
20.1 °C
26.8 °C
22.6 °C
20.4 °C
```

3 . Modifique el programa anterior para que soporte como entrada un archivo que no solo tiene la temperatura expresada en grados centígrados, sino que puede tener temperaturas expresadas en Fahrenheit o Kelvin, pero que siga generando un archivo de salida con las temperaturas en grados Fahrenheit. Para esto tenga en cuenta que la formula de conversión de Kelvin (°K) a Fahrenheit (°F) es la siguiente:

$$F = (9 * (K - 273,15)/5,0) + 32$$

Utilice el siguiente archivo de temperaturas de entradas:

```
50 °F
12.5 °C
287.35 °K
16.6 °C
291.35 °K
20.1 °C
299.95 °K
22.6 °F
20.4 °C
```

### Tabla de Contenidos

```
%%javascript
```

```
$.getScript('https://kmahelona.github.io/ipython_notebook_goodies/ipython_notebo
```

```
<IPython.core.display.Javascript object>
```

---

## Introducción a la Programación Orientada a Objetos

---

Python es un lenguaje de Programación Orientado a Objetos, lo que significa que puede manipular construcciones llamadas objetos. Se puede pensar en un objeto como una única estructura que contiene tanto datos como funciones, solo que las funciones en este contexto son llamadas *métodos*. En definitiva, los objetos son una manera de organizar datos y de relacionarlos con el código apropiado para manejarlo.

La Programación Orientada a Objetos introduce terminología, y una gran parte es simplemente darle un nuevo nombre a cosas que ya estuvimos usando.

Si bien Python nos provee un gran número de tipos ya definidos (int, float, str, dict, list, etc.), en muchas situaciones utilizar solamente estos tipos resultará insuficiente. En estas situaciones queremos poder crear nuestros propios tipos, que almacenen la información relevante para el problema a resolver y contengan las funciones para operar con esa información.

Supongamos un programa que gestiona jugadores de fútbol de un club, independientemente de los detalles de implementación, contar con un tipo de dato *jugador* que permita cargar los datos personales y profesionales nos brinda la posibilidad de tener un código mas legible y organizado. Por ejemplo, para cargar los datos de un nuevo jugador el código podría ser del siguiente modo:

```
pipa = Jugador('Lucas Alario', '8-10-1992', 'Delantero')
pipa.AgregarClub('Colon')
pipa.AgregarClub('River')
print("Club Actual: ", pipa.ClubActual())
```

Del ejemplo previo destacamos:

- `pipa = Jugador(...)` crea una nueva instancia de la clase `Jugador` y asigna este objeto a la variable local `pipa`, una estructura que contiene un conjunto de datos (nombre, fecha de nacimiento y posición) denominados atributos (o propiedades) y métodos (funciones asociadas al objeto)

### 8.1 Atributos y métodos

Veamos el modo de declarar este nuevo tipo `Jugador` con sus atributos y métodos.

```
class Jugador(object):
    """Clase Jugador"""
    def __init__(self, nombre=None, fechaNac=None, posicion=None):
        self.nombre = nombre
        self.fechaNac = fechaNac
        self.posicion = posicion
        self.clubes = []

    def setNuevoClub(self, club):
        '''agrega club a la lista de clubes'''
        self.clubes.append(club)

    def getClubActual(self):
        '''retorna último club'''
        return self.clubes[-1]

pipa = Jugador('Lucas Alario', '08-10-1992', 'Delantero')

pipa.setNuevoClub('Colon')
pipa.setNuevoClub('River')
print("Club Actual: ", pipa.getClubActual())

d10s = Jugador('El Diego', '30-10-1960', 'Enganche')
```

```
Club Actual: River
```

La clase anterior define la estructura de aquellos objetos que sean de tipo `Jugador()`. De los tres métodos que se observan, hay uno que merece especial atención:

- `__init__`: este método se denomina constructor, ya que está directamente asociado a la declaración e inicialización de un objeto. Esto es, en la el fragmento de código `pipa = Jugador('Lucas Alario', '8-10-1992', 'Delantero')` se lo invoca automáticamente. Los argumentos se corresponden con `nombre`, `fechaNac` y `posicion` respectivamente. El primer argumento, `self`, hace referencia al mismo objeto y es utilizado para definir sus atributos dentro del constructor.

Los métodos restantes no son más que funciones pertenecientes al objeto:

- `setNuevoClub()`: agrega un club donde jugó
- `getClubActual()`: retorna el último club

Los datos relativos al club se cargan en una lista almacenada en el atributo `clubes`. El uso de métodos para modificar atributos es denominado **encapsulamiento**. Es común encontrar métodos cuyos nombres empiecen con “set”, en aquellos casos donde los mismos realizan modificaciones sobre los datos, y métodos cuyos nombres empiezan con “get”, los cuales son utilizados para retornar datos. Esto es opcional, dado que podemos ponerle el nombre que se nos ocurra, pero es una buena costumbre llamarlos de este modo, al igual que el hecho de respetar el encapsulamiento (esto es, siempre modificar y obtener los datos, mediante el uso de un método propio del objeto).

## 8.2 Métodos especiales

Así como el constructor `__init__`, existen otros métodos especiales que, si están definidos en nuestra clase, Python los llamará por nosotros cuando se lo utilice en determinadas situaciones. Veamos algunos.

### 8.2.1 Impresión

Si está definido el método `__str__` dentro de la clase, entonces será invocado automáticamente cada vez que se utilice la función `print()` con el objeto como argumento. Veamos la implementación:

```
def __str__(self):
    salida = self.nombre
    salida += '\n' + '='*len(self.nombre) + '\n'
    salida += 'Club: ' + self.getClubActual() + '\n'
    salida += 'Posición: ' + self.posicion + '\n'
    return salida
```

Luego, al imprimirlo en pantalla obtendremos:

```
print(pipa)
```

```
Lucas Alario
=====
Club: River
Posición: Delantero
```

Esto incluso es equivalente a hacer

```
pipa.__str__()
```

```
Lucas Alario
=====
Club: River
Posición: Delantero
```

### 8.2.2 Comparación

Para resolver las comparaciones entre jugadores, será necesario definir algunos métodos especiales que permiten comparar objetos. En particular, cuando se quiere que los objetos puedan ser ordenados, los métodos que se debe definir son:

- `__lt__` menor que,
- `__le__` menor o igual,
- `__eq__` igual,
- `__ne__` distinto,

- `__gt__` mayor que,
- `__ge__` mayor o igual

Para dos objetos `x`, `y`:

- `x < y` llama a `x.__lt__(y)`,
- `x <= y` llama a `x.__le__(y)`,
- `x == y` llama a `x.__eq__(y)`,
- `x != y` llama a `x.__ne__(y)`,
- `x > y` llama a `x.__gt__(y)`,
- `x >= y` llama a `x.__ge__(y)`.

Para el ejemplo que estamos desarrollando, solamente programaremos el método `__lt__`, ya que al no ser un jugador menor que otro, nos retorna el complemento. En la comparación rearmaremos la fecha en el formato `aaaammdd` ya que al convertirla a un entero podremos compararla como un simple número, donde uno mas grande significa que el jugador es mas joven y, mas adulto, en caso contrario.

La implementación sería:

```
def __lt__(self, otro):
    '''si self es menor a otro'''
    dd1, mm1, aaaa1 = self.fechaNac.split('-')
    aaaammdd1 = aaaa1 + mm1 + dd1

    dd2, mm2, aaaa2 = otro.fechaNac.split('-')
    aaaammdd2 = aaaa2 + mm2 + dd2

    return (int(aaaammdd1) > int(aaaammdd2))
```

Luego, lo usamos:

```
d10s = Jugador('El Diego', '30-10-1960', 'Enganche')
print(pipa>d10s)
```

### 8.2.3 Algebraicos

Existen métodos especiales para todos los operadores matemáticos, de modo que al operar dos objetos, por ejemplo sumarlos, se invoca al método específico y se realiza la operación. Esto es también denominado sobrecarga de operadores, ya que se le asigna una función específica a un operador para un determinado objeto.

Para el ejemplo visto usaremos el monto del pase, así que agreguemos el atributo *valor* a la clase e incorporemos el método especial `__add__` de modo que al sumar objetos de tipo `Jugador()` se sumen estos campos.

```
def __add__(self, otro):
    return self.valor + otro.valor
```

Si ahora sumamos dos jugadores, obtendremos la suma de sus valores.

```
# asignamos valor a cada jugador
pipa.valor = 1130000
d10s.valor = 9000000

# sumamos dos jugadores
monto = pipa + d10s
print(monto)
```

Del mismo modo se implementan los métodos especiales para los siguientes operadores binarios

Operador	Método
+	<code>__add__(self, other)</code>
-	<code>__sub__(self, other)</code>
*	<code>__mul__(self, other)</code>
//	<code>__floordiv__(self, other)</code>
/	<code>__div__(self, other)</code>
%	<code>__mod__(self, other)</code>
**	<code>__pow__(self, other[, modulo])</code>
<<	<code>__lshift__(self, other)</code>
>>	<code>__rshift__(self, other)</code>
&	<code>__and__(self, other)</code>
^	<code>__xor__(self, other)</code>
	<code>__or__(self, other)</code>

Existen muchos otros métodos especiales como los de asignaciones extendidas y operadores unarios.

## 8.3 Herencia y polimorfismo

La herencia es un mecanismo de la programación orientada a objetos que sirve para crear clases nuevas a partir de otras preexistentes. Se heredan atributos y comportamientos y, partir de ella se crea una clase derivada con sus particularidades.

Por ejemplo, a partir de una clase `Jugador` podemos construir la clase `Capitan` que extiende a `Jugador` y agrega como atributo una lista de fechas de partidos que tuvo ese rol. Se puede ver como un caso particular de la clase `Jugador`, dado que tendrá los mismos atributos y métodos que un objeto de la clase `Jugador`, y a su vez tendrá algunos atributos y/o métodos extras.

Para indicar el nombre de la clase base, se la pone entre paréntesis a continuación del nombre de la clase. Veamos el modo de implementarla:

```
class Capitan(Jugador):
    "Clase que representa al capitan."

    def __init__(self, nombre=None, fechaNac=None, posicion=None, capitan=[]):
        "Constructor de Capitan"
```

```
# llama al constructor de Jugador
Jugador.__init__(self, nombre, fechaNac, posicion)
# nuevo atributo
self.capitan = capitan

def setCapitania(self, fecha):
    self.capitan.append(fecha)
```

En la implementación del método constructor (`__init__`) de `Capitan` se invoca al constructor de `Jugador`, luego, se agrega el atributo `capitan` y un método nuevo, `setCapitania`, que solamente existe en esta clase.

El hecho de heredar todas las características de la clase base hace que su uso sea prácticamente el mismo:

```
pulga = Capitan('Lionel Messi', '24-06-1987', 'Enganche')
pulga.setNuevoClub('Barcelona')
pulga.setCapitania('26-07-2008')
print(pulga)
```

Ahora bien, sería bueno diferenciar el método de impresión, ya que al imprimir en pantalla un jugador que es de tipo `Capitan`, muestre la última fecha de su capitania. Modificar un método heredado es lo que se denomina **Polimorfismo**. Veamos la diferencia:

```
def __str__(self):
    salida = Jugador.__str__(self)
    salida += 'Última capitania: ' + self.capitan[-1] + '\n'
    return salida
```

En la implementación del método `__str__` se invoca al de la clase base, y se agrega una línea más referida a la capitania.

El presente capítulo ha sido una introducción a la POO presentada en forma de tutorial, a continuación se expone el código completo de lo desarrollado durante la unidad.

```
class Jugador(object):
    """Clase Jugador"""
    def __init__(self, nombre=None, fechaNac=None, posicion=None, clubes=[], valor=None):
        self.nombre = nombre
        self.fechaNac = fechaNac
        self.posicion = posicion
        self.clubes = clubes
        self.valor = valor

    def setNuevoClub(self, club):
        '''agrega club a la lista de clubes'''
        self.clubes.append(club)

    def getClubActual(self):
        '''retorna último club'''
        return self.clubes[-1]
```



```

def __str__(self):
    salida = self.nombre
    salida += '\n' + '='*len(self.nombre) + '\n'
    salida += 'Club: ' + self.getClubActual() + '\n'
    salida += 'Posición: ' + self.posicion + '\n'
    return salida

def __lt__(self, otro):
    '''si self es menor a otro'''
    dd1, mm1, aaaa1 = self.fechaNac.split('-')
    aaaammdd1 = aaaa1 + mm1 + dd1

    dd2, mm2, aaaa2 = otro.fechaNac.split('-')
    aaaammdd2 = aaaa2 + mm2 + dd2

    return (int(aaaammdd1) > int(aaaammdd2))

def __add__(self, otro):
    return self.valor + otro.valor

class Capitan(Jugador):
    "Clase que representa al capitan."
    def __init__(self, nombre=None, fechaNac=None, posicion=None, capitan=[]):
        "Constructor de Capitan"
        # llama al constructor de Jugador
        Jugador.__init__(self, nombre, fechaNac, posicion)
        # nuevo atributo
        self.capitan = capitan
    def setCapitania(self, fecha):
        self.capitan.append(fecha)

    def __str__(self):
        '''sobreescribe la clase heredada'''
        salida = Jugador.__str__(self)
        salida += 'Última capitania: ' + self.capitan[-1] + '\n'
        return salida

pipa = Jugador('Lucas Alario', '08-10-1992', 'Delantero')
pipa.setNuevoClub('Colon')
pipa.setNuevoClub('River')
print(pipa)

d10s = Jugador('El Diego', '30-10-1960', 'Enganche')
d10s.setNuevoClub('Argentino Jr.')
d10s.setNuevoClub('Boca')
d10s.setNuevoClub('Barcelona')
d10s.setNuevoClub('Nápoles')
d10s.setNuevoClub('Sevilla')
d10s.setNuevoClub("Newell's")
d10s.setNuevoClub("Boca")

```

```
print(d10s)

pipa.valor = 1130000
d10s.valor = 9000000
monto = pipa + d10s

pulga = Capitan('Lionel Messi', '24-06-1987', 'Enganche')
pulga.setNuevoClub('Barcelona')
pulga.setCapitania('28-03-1981')
print(pulga)
```

```
Lucas Alario
=====
Club: River
Posición: Delantero

El Diego
=====
Club: Boca
Posición: Enganche

Lionel Messi
=====
Club: Barcelona
Posición: Enganche
Última capitania: 28-03-1981
```

Se recomienda profundizar este tema en el capítulo *Un primer vistazo a las clases* (pag. 61) del Tutorial de Python.

---

**Ejercicios**

---

todavía naranja che



---

## **Indices and tables**

---

- `genindex`
- `modindex`
- `search`