

---

# 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:





---

# 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 baasan 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 multiproposito, 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, intrucciones, 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 simmplemente en un programa que muestra en pantalla ese mensaje.

Renunciando a cualquier pretención 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 intrucción 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 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-15-3baa5c95d16e> 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('%0.2f %0.4f %0.3f' %(pi,pi,pi))
print('%4d' %uno)
```

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

<ipython-input-1-f45a2755e54d> in <module>()
----> 1 print('%0.2f %0.4f %0.3f' %(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 Operadores relacionales y lógicos

¿¿¿capaz va a la unidad siguiente!!! - Coincido :D

## 1.6.8 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.

### 1.6.9 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.

## 1.7 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')

('Diferencia de tiempo:', 1, 'hs', 1, 'min', 1, 'seg')
```





---

## 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**

### 2.1 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 pueden representar mediante el tipo int (de integer, entero) o el tipo long (largo). La única diferencia es que el tipo long permite almacenar números más grandes. Por ejemplo:

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

```
int
```

```
b = 28L
type(b)
```

```
long
```

#### 2.1.1 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
```

### 2.1.2 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
```

## 2.2 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 roben las ideas de otros"
type(b)
```

```
str
```

```
c = '''Un instrumento de poco costo y no más grande que un reloj, permitirá a su dueño
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
```

## 2.3 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
```

### 2.3.1 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 carácter a carácter, 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
```

## 2.3.2 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
```

Desglocemos 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>





---

## Estructuras de datos y control de flujo

---

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 por ejemplo que queremos hacer un programa que nos pregunte ciertas cosas, y en base a esto 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), determinarán 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).

### 3.1 Estructuras condicionales

La primer estructura de control que veremos son los condicionales, los cuales nos permiten comprobar condiciones y hacer que 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.

#### 3.1.1 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, 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:

```
if (temperatura >= 10) and (temperatura < 30):  
    print('Deberías ser amable con el medio ambiente e ir en bicicleta')
```

Esta sentencia se lee como: si (if) temperatura es menor o igual a 10, y temperatura es menor a 30, entonces ejecutar: `print('Deberías ser amable con el medio ambiente e ir en bicicleta')`. 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.

Una cuestión muy importante es asegurarnos de que el código está indentado tal cual se ha hecho en el ejemplo, es decir, asegurarnos de pulsar Tabulación en la línea que está debajo de los 2 puntos (:), dado que esta es la forma de Python de saber que nuestra intención es que las sentencias que están indentadas, se ejecute sólo en el caso de que se cumpla la condición.

### 3.1.2 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('Deberías ser amable con el medio ambiente e ir en bicicleta')
else:
    print('La temperatura no es agradable, te recomiendo ir en auto.')
```

Esto se lee como *si temperatura es mayor o igual a 10 y temperatura es menor que 30, entonces mostrar el mensaje 'Deberías ser amable con el medio ambiente e ir en bicicleta', sino mostrar el mensaje 'La temperatura no es agradable, te recomiendo ir en auto.'* 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).

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

if (temperatura >= 10) and (temperatura < 30):
    print('Deberías ser amable con el medio ambiente e ir en bicicleta')
else:
    print('La temperatura no es agradable, le recomiendo ir en auto')

print('Que tenga buen día!')
```

Ingrese la temperatura en °C:21

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

<ipython-input-1-286b3e307026> in <module>()
      1 temperatura = input('Ingrese la temperatura en °C:')
      2
----> 3 if (temperatura >= 10) and (temperatura < 30):
      4     print('Deberías ser amable con el medio ambiente e ir en bicicleta')
```

```
5 else:
```

```
TypeError: unorderable types: str() >= int()
```

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 están dentro del *if* y el *else*, lo restante se seguirá ejecutando de manera secuencial.

### 3.1.3 Estructuras anidadas

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 determinada 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 = input('Ingrese la temperatura en °C:')
distancia = input('Ingrese la distancia a recorrer en km:')

if (temperatura >= 10) and (temperatura < 30):
    if (distancia <= 15):
        print('Deberías ser amable con el medio ambiente e ir en bicicleta')
    else:
        print('El clima es agradable, pero la distancia es muy larga. Le recomie
else:
    print('La temperatura no es agradable, le recomiendo ir en auto.')

print('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 ‘Deberías ser amable con el medio ambiente e ir en bicicleta’, en caso contrario, se muestra el mensaje ‘El clima es agradable, pero la distancia es muy larga. Le recomiendo ir en auto’. Por otro lado, si el valor de la variable *temperatura* no está 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*.

## **3.2 Estructuras repetitivas**

### **3.2.1 Estructura repetitiva *while***

### **3.2.2 Estructura repetitiva *for***

## **3.3 Estructura de datos *listas y diccionarios***

### **3.3.1 Listas**

### **3.3.2 Diccionarios**

## **3.4 Manipulando textos con strings**

---

## Funciones, archivos, diccionarios

---

### 4.1 Definiendo funciones

#### 4.1.1 Variables globales y locales

Agrupando el código en módulos

### 4.2 Números aleatorios

### 4.3 Lectura y escritura de archivos

### 4.4 Diccionarios



---

**Clases y objetos**

---





---

## Indices and tables

---

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