EUGENIA BAHIT

INTRODUCCIÓN AL LENGUAJE PYTHON

MATERIAL DE ESTUDIO

Informes e inscripción:
Curso: http://escuela.eugeniabahit.com | **Certificaciones**: http://python.laeci.org

INTRODUCCIÓN AL LENGUAJE PYTHON

Libro de estudio del curso de la Escuela de Informática Eugenia Bahit

SUMARIO

PRIMER ACERCAMIENTO AL SCRIPTING

En programación, un *script* es un archivo de código fuente con instrucciones sencillas, que puede ser ejecutado a través de la línea de comandos. Se conoce como *scripting* a la **técnica de programación** empleada para crear este tipo de archivos.

Para que un archivo de código fuente sea considerado un script, debe cumplir las siguientes características:

1. Ser ejecutable.
2. Estar escrito en un lenguaje que pueda ser interpretado por el ordenador.
3. No depender de otros archivos.

1. Archivo ejecutable

Para que un archivo pueda ser ejecutado, hay que otorgarle permisos de ejecución:

```
chmod +x nombre-del-archivo
```

2. Lenguaje interpretado

Un *script* puede ser escrito en cualquier lenguaje interpretado, soportado por el Sistema Operativo, como Python, Perl, GNU Bash, PHP, Ruby, etc.

Para que el sistema operativo sepa qué intérprete utilizar para entender y ejecutar el *script*, se debe colocar la ruta de dicho intérprete en la primera línea del archivo, antecedida por los símbolos **#!**

Lo anterior se conoce como **hashbang** o **shebang**.

Dado que la ruta de un intérprete puede ser distinta en los diversos Sistemas Operativo o distribuciones, realizar la llamada al intérprete a través de **env**, suele ser una práctica habitual:

```
#!/usr/bin/env python
```

Por este motivo, un *script* no requiere de una extensión para poder ser interpretado.

Algunos ejemplos:

`#!/usr/bin/env python`	Intérprete Python (versión por defecto del sistema)
`#!/usr/bin/env python2`	Fuerza a utilizar el intérprete de Python 2.x
`#!/usr/bin/env python3`	Fuerza a utilizar el intérprete de Python 3.x
`#!/usr/bin/env bash`	Intérprete de GNU Bash

3. Script vs Programa

Si un archivo de código fuente cumple es ejecutable, contiene la ruta del intérprete pero depende de otros archivos, entonces se lo considera un programa.

CONVERTIR UN SCRIPT EN COMANDO DEL SISTEMA

Archivo: *hola-mundo.py*

```
#!/usr/bin/env python2
print "Hola Mundo!"
```

Crear un archivo llamado «*hola-mundo.py*» (con el contenido anterior), asignarle permisos de ejecución y probar el script en la línea de comandos ejecutando la siguiente instrucción:

```
./hola-mundo.py
```

Ahora, como root, copiar el archivo a la carpeta */usr/bin* con el nombre «*hola-mundo*»:

```
cp hola-mundo.py /usr/bin/hola-mundo
```

El *script* está ahora disponible para ser ejecutado como comando.

```
usuario@equipo:~$ hola-mundo
Hola Mundo!
usuario@equipo:~$
```

ACERCA DE PYTHON

Dentro de los **lenguajes informáticos**, Python, pertenece al grupo de los **lenguajes de programación** y puede ser clasificado como un **lenguaje interpretado**, **de alto nivel**, **multiplataforma**, de **tipado dinámico** y **multiparadigma**.

Para la escritura de código, y a fin de alcanzar un mecanismo estándar en la forma de programar, **provee unas reglas de estilo** definidas a través de la *Python Enhancement Proposal N° 8 (PEP 8)* , que serán expuestas a lo largo de todo el libro.

GLOSARIO

Lenguaje informático: es un idioma artificial utilizado por ordenadores, cuyo fin es transmitir información de algo a alguien. Los lenguajes informáticos, pueden clasificarse en: a) lenguajes de programación (Python, PHP, Pearl, C, etc.); b) lenguajes de especificación (UML); c) lenguajes de consulta (SQL); d) lenguajes de marcas (HTML, XML); e) lenguajes de transformación (XSLT); f) protocolos de comunicaciones (HTTP, FTP); entre otros.

Lenguaje de programación: es un lenguaje informático, diseñado para expresar órdenes e instrucciones precisas, que deben ser llevadas a cabo por una computadora. El mismo puede utilizarse para crear programas que controlen el comportamiento físico o lógico de un ordenador. Está compuesto por una serie de símbolos, reglas sintácticas y semánticas que definen la estructura del lenguaje.

Lenguajes de alto nivel: son aquellos cuya característica principal, consiste en una estructura sintáctica y semántica legible, acorde a las capacidades cognitivas humanas. A diferencia de los lenguajes de bajo nivel, son independientes de la arquitectura del hardware, motivo por el cual, asumen mayor portabilidad.

Lenguajes interpretados: a diferencia de los compilados, no requieren de compiladores para ser ejecutados, sino de intérpretes. Un intérprete, actúa de manera similar a un compilador, con la salvedad de que ejecuta el programa directamente, sin necesidad de generar previamente un archivo ejecutable. Ejemplo de lenguajes de programación interpretados son Python, PHP, Ruby, Lisp, entre otros.

Tipado dinámico: un lenguaje de tipado dinámico es aquel cuyas variables, no requieren ser definidas asignando su tipo de datos, sino que éste, se auto-asigna en tiempo de ejecución, según el valor declarado.

Multiplataforma: significa que puede ser interpretado en diversos Sistemas Operativos como GNU/Linux, OpenBSD, sistemas privativos, entre otros.

Multiparadigma: acepta diferentes paradigmas (técnicas) de programación, tales como la orientación a objetos, la programación imperativa y funcional.

Código fuente: es un conjunto de instrucciones y órdenes lógicas, compuestos de algoritmos que se encuentran escritos en un determinado lenguaje de programación, las cuales deben ser interpretadas o compiladas, para permitir la ejecución de un programa informático.

ELEMENTOS DEL LENGUAJE

Python, al igual que todo lenguaje de programación, se compone de una serie de elementos y estructuras que componen su sintaxis.

A continuación, se abarcarán los principales elementos.

VARIABLES

Una variable es un espacio para almacenar datos modificables, en la memoria del ordenador.

En Python, una variable se define con la sintaxis:

```
nombre_de_la_variable = valor_de_la_variable
```

Cada variable, tiene un nombre y un valor. Este último, define el tipo de datos de la variable.

Existe un tipo de espacio de almacenamiento denominado **constante**. Una constante, se utiliza para definir valores fijos, que no requieran ser modificados. En Python, el concepto de constante es simbólico ya que todo espacio es variable.

PEP 8: variables

Utilizar nombres descriptivos y en minúsculas. Para nombres compuestos, separar las palabras por guiones bajos. Antes y después del signo =, debe haber uno (y solo un) espacio en blanco

Correcto: `mi_variable = 12`

Incorrecto: `MiVariable = 12 | mivariable = 12 | mi_variable=12`

PEP 8: constantes

Utilizar nombres descriptivos y en mayúsculas separando palabras por guiones bajos.

Ejemplo: `MI_CONSTANTE = 12`

ENTRADA Y SALIDA

Para **imprimir un valor en pantalla**, hasta la versión 2.7 de Python, se utiliza la palabra clave `print`:

```
mi_variable = 15
print mi_variable
```

A partir de la versión 3.0 de Python, se utiliza la función `print()`:

```
mi_variable = 15
print(mi_variable)
```

Lo anterior, en ambos casos, imprime el valor de la variable *mi_variable* en pantalla.

Para **obtener datos solicitados al usuario**, hasta la versión 2.7 de Python, se utiliza la función `raw_input()`:

```
mi_variable = raw_input("Ingresar un valor: ")
```

A partir de la versión 3.0 de Python, se utiliza la función `input()`, también disponible en versiones anteriores, pero con un funcionamiento diferente:

```
mi_variable = input("Ingresar un valor: ")
```

TIPOS DE DATOS

Una variable puede contener valores de diversos tipos. Estos tipos se listas a continuación.

Cadena de texto (string):

```
mi_cadena = "Hola Mundo!"
otra_cadena = 'Hola Mundo!'

mi_cadena_multilinea = """
Esta es una cadena
de varias lineas
"""
```

Número entero:

```
edad = 35
```

Número real:

```
precio = 35.05
```

Booleano (verdadero / Falso):

```
verdadero = True
falso = False
```

Existen además, otros tipos de datos más complejos, que se abarcarán más adelante.

OPERADORES ARITMÉTICOS

Un operador aritmético es aquel que permite realizar operaciones aritméticas sobre las variables.

Símbolo	Significado	Ejemplo	Resultado
+	Suma	a = 10 + 5	a es 15
-	Resta	a = 12 - 7	a es 5
-	Negación	a = -5	a es -5
*	Multiplicación	a = 7 * 5	a es 35
**	Exponente	a = 2 ** 3	a es 8
/	División	a = 12.5 / 2	a es 6.25
//	División entera	a = 12.5 // 2	a es 6.0
%	Módulo	a = 27 % 4	a es 3

PEP 8: operadores

Siempre colocar un espacio en blanco, antes y después de un operador

Un ejemplo sencillo con variables y operadores aritméticos:

```
monto_bruto = 175
tasa_interes = 12.5
monto_interes = monto_bruto * tasa_interes / 100
tasa_bonificacion = 5.0
importe_bonificacion = monto_bruto * tasa_bonificacion / 100
monto_neto = (monto_bruto - importe_bonificacion) + monto_interes
```

COMENTARIOS

En un archivo de código fuente, un comentario es un texto explicativo que no es procesado por el intérprete y solo tiene una utilidad humana (no informática). Los comentarios pueden ser de una o varias líneas:

```
# Esto es un comentario de una sola línea
mi_variable = 15

"""Y este es un comentario
de varias líneas"""
mi_variable = 15
mi_variable = 15  # Este es un comentario en línea
```

En los comentarios, pueden incluirse palabras que representen acciones:

```
# TODO esto es algo por hacer
# FIXME esto es algo que debe corregirse (un fallo)
# XXX esto es algo que debe corregirse (error crítico)
```

PEP 8: comentarios

Comentarios en la misma línea del código deben separarse con dos espacios en blanco. Luego del símbolo # debe ir un solo espacio en blanco.

Correcto: `a = 15  # Edad de María`

Incorrecto: `a = 15 # Edad de María`

CODIFICACIÓN DE CARACTERES (ENCODING)

En versiones anteriores a Python 3.0, la especificación de la codificación de caracteres a emplearse (cuando ésta no es ASCII), resulta obligatoria.

A partir de la versión 3.0, esta declaración es opcional.

```
# -*- coding: utf-8 -*-
```

utf-8 podría ser cualquier otra codificación de caracteres. Por defecto, versiones anteriores a Python 3, utilizan la codificación ASCII como predeterminada. En caso de emplear caracteres no ASCII en versiones anteriores a la 3.0, Python generará un error:

```
variable = "En el Ñágara encontré un Ñandú"
SyntaxError: Non-ASCII character[...]
```

Indicando el *encoding* adecuado, el archivo se ejecutará con éxito:

```
# -*- coding: utf-8 -*-

variable = "En el Ñágara encontré un Ñandú"
```

TIPOS DE DATOS COMPLEJOS

Python, posee además de los tipos ya vistos, 3 tipos más complejos, que admiten una colección de datos. Estos tipos son:

- Tuplas
- Listas
- Diccionarios

Estos tres tipos, pueden almacenar colecciones de datos de diversos tipos y se diferencian por su sintaxis y por la forma en la cual los datos pueden ser manipulados.

TUPLAS

Una tupla es una **variable que permite almacenar varios datos inmutables** (no pueden ser modificados una vez creados), y estos datos pueden ser de tipos diferentes:

```
mi_tupla = ('cadena de texto', 15, 2.8, 'otro dato', 25)
```

Se puede acceder a cada uno de los datos mediante su índice correspondiente. El índice se corresponde con la posición del elemento en la colección, siendo 0 (cero), el índice del primer elemento:

```
mi_tupla[1]  # Salida: 15
```

También se puede acceder a una porción de la tupla, indicando (opcionalmente) desde el índice de inicio hasta el índice de fin:

```
mi_tupla[1:4]  # Devuelve: (15, 2.8, 'otro dato')
mi_tupla[3:]   # Devuelve: ('otro dato', 25)
mi_tupla[:2]   # Devuelve: ('cadena de texto', 15)
```

Otra forma de acceder a la tupla de forma inversa (de atrás hacia adelante), es colocando un índice negativo:

```
mi_tupla[-1]  # Salida: 25
mi_tupla[-2]  # Salida: otro dato
```

LISTAS

Una lista es similar a una tupla en todos los aspectos. La diferencia radica en que los elementos de una lista sí pueden ser modificados:

```
mi_lista = ['cadena de texto', 15, 2.8, 'otro dato', 25]
```

A las listas se accede igual que a las tuplas, por su número de índice:

```
mi_lista[1]    # Salida: 15
mi_lista[1:4]  # Devuelve: [15, 2.8, 'otro dato']
mi_lista[-2]   # Salida: otro dato
```

Las lista no son inmutables: permiten modificar los datos una vez creados:

```
mi_lista[2] = 3.8  # el tercer elemento ahora es 3.8
```

Al poder ser modificadas, las listas, a diferencia de las tuplas, permiten agregar nuevos valores:

```
mi_lista.append('Nuevo Dato')
```

DICCIONARIOS

Los diccionarios, al igual que las tuplas y las listas, son colecciones. La diferencia con estos, es que mientras que los elementos de las listas y las tuplas se asocian a un número de índice (o posición), los valores de un diccionario se asocian a un nombre de clave:

```
mi_diccionario = {
    'clave_1': valor_1,
    'clave_2': valor_2,
    'clave_7': valor_7
}
mi_diccionario['clave_2']  # Salida: valor_2
```

Un diccionario permite eliminar cualquier entrada:

```
del(mi_diccionario['clave_2'])
```

Al igual que las listas, el diccionario permite modificar los valores:

```
mi_diccionario['clave_1'] = 'Nuevo Valor'
```

Y también es posible agregar nuevos elementos, asignando valores a nuevas claves:

```
mi_diccionario['nueva_clave'] = 'nuevo elemento'
```

ESTRUCTURAS DE CONTROL DE FLUJO

Una estructura de control es un bloque de código fuente que permite agrupar instrucciones de forma controlada. A continuación se describirán dos tipos de estructuras de control:

1. **Estructuras de control condicionales:** controlan el flujo de los datos a partir de condiciones.
2. **Estructuras de control iterativas:** controlan el flujo de los datos, ejecutando una misma acción de forma repetida.

IDENTACIÓN

En Python, las estructuras de control se delimitan sobre la base de bloques de código identados. En un lenguaje informático, se llama identación al sangrado del código fuente.

No todos los lenguajes de programación, necesitan de una identación, aunque sí se estila implementarla, a fin de otorgar mayor legibilidad al código fuente. **En el caso de Python, la identación es obligatoria**, ya que de ella, dependerá su estructura y la forma de controlar la información.

PEP 8: identación

Una identación de **4 (cuatro) espacios en blanco**, indicará que las instrucciones identadas, forman parte de una misma estructura de control.

Una estructura de control, entonces, se define de la siguiente forma:

```
inicio de la estructura de control:
    expresiones
```

ESTRUCTURAS DE CONTROL DE FLUJO CONDICIONALES

Las estructuras de control condicionales son aquellas que permiten evaluar si una o más condiciones se cumplen, para decir qué acción ejecutar.

Las condiciones se evalúan como verdaderas o falsas. O la condición se cumple (la condición es verdadera), o la condición no se cumple (la condición es falsa).

En la vida diaria, toda persona actúa de acuerdo a la evaluación de condiciones, solo que no siempre se hace de forma explícita o evidente.

Por ejemplo:

> **Si** el semáforo está en verde, cruzar la calle. **Sino**, esperar a que el semáforo se ponga en verde.

En ocasiones, se evalúa más de una condición para llegar a ejecutar la misma acción:

> **Si** el semáforo está en verde **o** no hay vehículos circulando, cruzar la calle. **Sino**, esperar a que el semáforo se ponga en verde.

Para describir la evaluación a realizar sobre una condición, se utilizan **operadores relacionales** o de **comparación**:

Símbolo	Significado	Ejemplo	Resultado
==	Igual que	5 == 7	Falso
!=	Distinto que	rojo != verde	Verdadero

<	Menor que	8 < 12	Verdadero
>	Mayor que	7 > 12	Falso
<=	Menor o igual que	12 <= 12	Verdadero
>=	Mayor o igual que	4 >= 5	Falso

Para evaluar más de una condición simultáneamente, se utilizan **operadores lógicos**:

Operador	**Ejemplo**	**Resultado***	
and (y)	5 == 7 and 7 < 12	0 y 0	Falso
	9 < 12 and 12 > 7	1 y 1	Verdadero
	9 < 12 and 12 > 15	1 y 0	Falso
or (o)	12 == 12 or 15 < 7	1 o 0	Verdadero
	7 > 5 or 9 < 12	1 o 1	Verdadero
xor	4 == 4 xor 9 > 3	1 o 1	Falso
(o excluyente)	4 == 4 xor 9 < 3	1 o 0	Verdadero

(*) 1 indica resultado verdadero de la condición, mientras que 0, indica falso.

Las estructuras de control de flujo condicionales, se definen mediante el uso de tres palabras claves reservadas, del lenguaje: **`if`** (si), **`elif`** (sino, si) y **`else`** (sino).

Algunos ejemplos:

Si semáforo está en verde, cruzar. Sino, esperar.

```
if semaforo == verde:
    cruzar()
else:
    esperar()
```

En el ejemplo anterior, *cruzar* y *esperar* son dos acciones (verbos). Los verbos o acciones, en programación, se denominan funciones y sintácticamente, se las diferencia de las variables por estar acompañadas de un par de paréntesis. Las funciones serán abarcadas más adelante.

Si gasto hasta $100, pago con dinero en efectivo. Sino, si gasto más de $100 pero menos de $300, pago con tarjeta de débito. Sino, pago con tarjeta de crédito.

```
if gasto <= 100:
    pagar_en_efectivo()
elif gasto > 100 and gasto < 300:
    pagar_con_debito()
else:
    pagar_con_credito()
```

Si la compra es mayor a $100, obtengo un descuento del 10%

```
descuento = 0

if compra > 100:
    descuento = compra * 0.10
```

ESTRUCTURAS DE CONTROL ITERATIVAS

Las estructuras de control iterativas (también llamadas cíclicas o bucles), permiten ejecutar un mismo código, de forma repetida, *n* cantidad de veces mientras se cumpla una condición.

Python dispone dos estructuras de control iterativas:

- El bucle **while**
- El bucle **for**

BUCLE WHILE

Este bucle se encarga de ejecutar una misma acción *mientras que* una determinada condición se cumpla:

> Mientras que año sea menor o igual a 2012, imprimir la frase "Informes de *<año>*".

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

year = 2001

while year <= 2012:
    print "Informes de", year
    year += 1
```

La iteración anterior, generará la siguiente salida:

```
Informes de 2001
Informes de 2002
Informes de 2003
Informes de 2004
Informes de 2005
Informes de 2006
Informes de 2007
Informes de 2008
Informes de 2009
Informes de 2010
Informes de 2011
Informes de 2012
```

Observar la última línea:

```
year += 1
```

En cada iteración se incrementa en 1 (uno) el valor de la variable `year` que condiciona el bucle . Si no se incrementase esta variable, su valor siempre sería igual a `2001` y por lo tanto, el bucle se ejecutaría de forma infinita, ya que la condición `year <= 2012` siempre se estaría cumpliendo.

Notación BNF (Bakus-Naur Form)

Es un formato utilizado en programación e informática en general, para documentar expresiones lógicas. El uso de:

<nombre>

indica que donde aparece <nombre> deberá reemplazarse dicho modificador, por el valor de un elemento que represente un nombre. Por ejemplo:

`<Apellido>, <Nombre>`

Podría reemplazarse por:

`Bahit, Eugenia`

Romper un bucle

Cuando el valor que condiciona la iteración no puede incrementarse, puede romperse el bucle mediante la instrucción **break** precedida de un condicional:

```
while True:
    nombre = raw_input("Indique su nombre: ")
    if nombre:
       break
```

El bucle anterior, incluye un condicional anidado que verifica si la variable nombre es verdadera (solo será verdadera si el usuario aporta un dato cuando le es solicitado en pantalla). Si es verdadera, el bucle se rompe (*break*). Sino, seguirá ejecutándose hasta que el usuario ingrese un dato cuando le es solicitado.

BUCLE FOR

El bucle **for** de Python, es aquél que facilita la iteración sobre los elementos de una tupla o lista. El bucle for siempre se utilizará sobre una lista o tupla, de forma tal que en cada iteración, se puedan ejecutar las misma acciones sobre cada uno de los elementos de la lista o tupla.

Por cada elemento en mi_lista, imprimir el elemento

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

mi_lista = ['Juan', 'Antonio', 'Pedro', 'Ana']

for elemento in mi_lista:
    print(elemento)
```

Por cada color en mi_tupla, imprimir color

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

mi_tupla = ('rosa', 'verde', 'celeste', 'amarillo')

for color in mi_tupla:
    print(color)
```

En los ejemplos anteriores, *elemento* y *color*, son dos **variables declaradas en tiempo de ejecución** (es decir, se declaran dinámicamente durante el bucle y se sobrescriben en cada iteración), asumiendo como valor, el de cada elemento de la lista (o tupla) en cada iteración.

Otra forma de iterar con el bucle **for**, puede emular a *while*:

> Por cada año en el rango 2001 a 2013, imprimir la frase "Informes de <*año*>".

```
for year in range(2001, 2013):
    print "Informes de", year
```

FUNCIONES

Una función es una forma de agrupar expresiones y algoritmos de forma tal que estos queden contenidos dentro de «una cápsula», que solo pueda ejecutarse cuando el programador la invoque. Una vez que una función es definida, puede ser invocada tantas veces como sea necesario.

Funciones incorporadas

Una función puede ser provista por el lenguaje o definida por el usuario. A las funciones provistas por el lenguaje se las denomina «funciones incorporadas» y en inglés se conocen como «*Built-in functions*».

FUNCIONES DEFINIDAS POR EL USUARIO

En Python la definición de funciones se realiza mediante la instrucción **`def`** más un nombre de función descriptivo -para el cuál, aplican las mismas reglas que para el nombre de las variables- seguido de paréntesis de apertura y cierre.

Como toda estructura de control en Python, la definición de la función finaliza con dos puntos (:) y el algoritmo que la compone, irá identado con 4 espacios en blanco:

```
def mi_funcion():
    # aquí el algoritmo identado
```

Una función no es ejecutada hasta tanto no sea invocada. Para invocar una función, simplemente se la llama por su nombre:

```
def mi_funcion():
    print "Hola Mundo"

funcion()
```

Las funciones pueden **retornar datos**:

```
def funcion():
    return "Hola Mundo"
```

Y el valor de retorno de una función puede **almacenarse** en una variable:

```
frase = funcion()
```

Imprimirse:

```
print funcion()
```

O **ignorarse**:

```
funcion()
```

SOBRE LOS PARÁMETROS

Un parámetro es un valor que la función espera recibir cuando sea llamada (invocada). Una función puede esperar uno o más parámetros (que son definidos entre los paréntesis y separados por una coma) o ninguno.

```
def mi_funcion(param1, param2):
    pass
```

> La instrucción **pass** se utiliza para completar una estructura de control que no realiza ninguna acción.

Los parámetros que una función espera, serán utilizados por ésta, dentro de su algoritmo, a modo de **variables de ámbito local**. Es decir, que los parámetros serán variables locales que solo son accesibles dentro de la función:

```
def calcular_neto(bruto, alicuota):
    iva = bruto * float(alicuota) / 100
    neto = bruto + iva
    return neto
```

Si se quisiera acceder a esas variables locales fuera del ámbito de la función, se obtendría un error:

```
def calcular_neto(bruto, alicuota):
    iva = bruto * float(alicuota) / 100
    neto = bruto + iva
    return neto

print bruto  # Retornará el error: NameError: name 'bruto' is not defined
```

Al llamar a una función, **se le deben pasar sus argumentos en el mismo orden en el que los espera**. Para evitar pasarlos en el mismo orden, pueden utilizarse claves como argumentos (definidos más abajo).

PARÁMETROS POR OMISIÓN

Es posible asignar valores por defecto a los parámetros de las funciones. Esto significa, que la función podrá ser llamada con menos argumentos de los que ha definido:

```
def calcular_neto(bruto, alicuota=21):
    iva = bruto * float(alicuota) / 100
    neto = bruto + iva
    return neto

calcular_neto(100)        # Retorna 121.0
calcular_neto(100, 10.5)  # Retorna 110.5
```

PEP 8: Funciones

A la definición de una función la deben anteceder dos líneas en blanco.

Al asignar parámetros por omisión, no debe dejarse espacios en blanco ni antes ni después del signo =.

Los parámetros por omisión deben ser definidos a continuación de los parámetros obligatorios.

CLAVES COMO ARGUMENTOS

Las claves como argumentos *(keword arguments)* son una característica de Python que no todos los lenguajes poseen. Python permite llamar a una función pasándole los argumentos esperados como pares de `claves=valor`:

```
def funcion(obligatorio1, opcional='valor opcional', opcional_dos=15):
    pass

funcion('valor obligatorio', opcional_dos=43)
```

PARÁMETROS ARBITRARIOS

Es posible que una función espere recibir un número arbitrario -desconocido- de argumentos. Estos argumentos, llegarán a la función en forma de tupla.

Para definir argumentos arbitrarios en una función, se antecede un asterisco (*) al nombre del parámetro :

```
def funcion(obligatorio, *arbitrarios):
    pass


funcion('fijo', 1, 2, 3, 4, 5)
funcion('fijo', 1, 2)
funcion('fijo', 1, 2, 3, 4, 5, 6, 7)
funcion('fijo')
```

Cuando una función espera recibir parámetros obligatorios y arbitrarios, **los arbitrarios siempre deben suceder a los obligatorios**.

Es posible también, obtener parámetros arbitrarios como pares de clave=valor. En estos casos, al nombre del parámetro deben precederlo dos asteriscos (**):

```
def funcion(obligatorio, **arbitrarios):
    pass


funcion('fijo', a=1, b=2, c=3)
```

El **recorrido de los parámetros arbitrarios** es como el recorrido de cualquier tupla (para parámetros arbitrarios sin clave) o de cualquier diccionario (para los parámetros arbitrarios con clave):

```
def funcion(*arbitrarios):
    for argumento in arbitrarios:
       pass


def funcion(**arbitrarios):
    for argumento in arbitrarios:
       valor = arbitrarios[argumento]
```

DESEMPAQUETADO DE PARÁMETROS

Al invocar a una función se le pueden pasar los parámetros que espera dentro de una lista o de un diccionario (donde los nombres de las claves equivalen al nombre de los argumentos). A este procedimiento se lo conoce como desempaquetado de parámetros:

```
def funcion(uno, dos, tres):
    pass


# DESEMPAQUETADO DE LISTAS
parametros = [1, 2, 3]
calcular(*parametros)

# DESEMPAQUETADO DE DICCIONARIOS
```

```
parametros = dict(uno=1, dos=2, tres=3)
calcular(**parametros)

parametros = {'uno': 1, 'dos': 2, 'tres': 3}
calcular(**parametros)
```

LLAMADAS RECURSIVAS Y DE RETORNO

Una función puede llamar a otra función que retorne un valor. Esto se conoce como llamada de retorno:

```
def retornar(algo):
    return str(algo)


def llamar():
    algo = retornar()
```

La llamada interna a otra función puede almacenarse, retornarse o ignorarse.

```
def almacenar():
    algo = retornar()


def volver_a_retornar():
    return retornar()


def ignorar():
    retornar()
```

Cuando la llama que se hace es a la misma función que se llama, se conoce como **llamada recursiva**.

```
def get_nombre():
    nombre = raw_input("Nombre: ")
    if not nombre:
        get_nombre()
```

SOBRE LA FINALIDAD DE LAS FUNCIONES

Una función puede tener cualquier tipo de algoritmo y cualquier cantidad de instrucciones. Sin embargo, **una buena práctica** indica que la finalidad de una función, debe ser **realizar una única acción**.

INYECCIÓN DE VARIABLES

Una variable se «inyecta» en una cadena de texto haciendo que su valor pase a formar parte de la cadena. Esto se hace mediante una operación de **formato**. Esto es necesario cuando la cadena a ser inyectada debe contener datos que son variables.

Para inyectar variables dentro de cadenas, las cadenas deben ser preparadas mediante el uso de modificadores. Un **modificador** puede ser un par de llaves vacías {} o un par de llaves con nombre {nombre}.

```
cadena = "Cadena preparada para recibir dos datos variables: {} y {}."
```

La misma cadena con modificadores con nombre:

```
cadena = "Cadena preparada para recibir dos datos variables: {dato1} y {dato2}."
```

Luego, se da formato a la cadena pasando una lista de variables que serán enlazadas a cada modificador:

```
cadena = "Cadena preparada para recibir dos datos variables: {} y {}."
resultado = cadena.format(variable1, variable2)
```

Y si tiene modificadores con nombre:

```
cadena = "Cadena preparada para recibir dos datos variables: {dato1} y {dato2}."
resultado = cadena.format(dato1=variable1, dato2=variable2)
```

La función **format()** es un método del objeto string. Los métodos son funciones. Los objetos, variables de un tipo de datos más complejo. Todo esto será abarcado en profundidad, en los cursos de *Scripting*.

IMPORTACIÓN DE MÓDULOS

Módulo. En Python se considera módulo a cualquier archivo .py.

Paquete. En Python se considera paquete es una carpeta que contiene módulos, y un archivo `__init__.py` que puede estar (o no) vacío.

```
.
└── paquete
    ├── __init__.py
    ├── modulo1.py
    ├── modulo2.py
    └── modulo3.py
```

Espacio de nombres

La importación de módulos y paquetes se realiza a través del espacio de nombres, el cual estará determinado por la ruta del archivo importar (se omiten las extensiones .py y las barras diagonales son sustituidas por un punto), tal que si la ruta de un archivo es *foo/bar/baz.py* su espacio de nombre será *foo.bar.baz*

Ruta de inclusión

Python, en primer lugar buscará los archivos en las rutas de importación propias, y en segundo lugar, en el directorio desde el cual se esté ejecutando el script.

Importación total y parcial

Importar un módulo significa incluir el contenido de un archivo dentro de otro.

Se pueden importar módulos completos o solo elementos parciales como variables, funciones, etc.

Para importar un módulo completo se utiliza la palabra clave *import* mientras que para importar elementos se utiliza la dupla *from / import*:

```
import modulo
import paquete.modulo
import paquete.subpaquete.modulo

from modulo import variable
from modulo import variable, funcion
from modulo import *
```

El **asterisco** equivale a importar todos los elementos contenidos en un módulo aunque no es igual que importar el módulo completo. Su uso está **desaconsejado** en las **PEP 8**.

```
import modulo         # para llamar a 'A' dentro de modulo: modulo.A
from modulo import *  # para llamar a 'A' dentro de modulo: A
```

Los módulos a importar pueden ser archivos propios (en ese caso, se convierte el *script* en un programa) o **módulos de la librería estándar** de Python (abarcados en el curso de *Scripting*).

PEP 8: importación

La importación de módulos debe realizarse al comienzo del documento, en orden alfabético de paquetes y módulos.

Primero deben importarse los módulos propios de Python. Luego, los módulos de terceros y finalmente, los módulos propios de la aplicación.

Entre cada bloque de imports, debe dejarse una línea en blanco.

Uso de elementos importados

Tanto si se ha importado un módulo completo como elementos independientes, se accede a los elementos de dicho módulo a través del espacio de nombre importado:

```
import modulosypaquetes.paquete
modulosypaquetes.paquete.modulo.funcion()

import modulosypaquetes.paquete.modulo
modulosypaquetes.paquete.modulo.funcion()

from modulosypaquetes.paquete import modulo
modulo.funcion()

from modulosypaquetes.paquete.modulo import funcion
funcion()
```

Alias

Es posible crear alias (en tiempo de importación) para acceder a los espacios de nombre de forma abreviada:

```
import modulosypaquetes.paquete.modulo as m
m.funcion()
```

PYTHON 2 VS PYTHON 3

Contexto científico. Introducción.

Como se comentó al comienzo de este documento, Python es un lenguaje de programación. Esto implica que como tal, ha sido concebido a partir de un grupo de principios lógicos y teorías científicas.

Por lo tanto, es válido afirmar que:

> Un lenguaje de programación es una forma de aplicar un conjunto de principios y teorías en un contexto práctico.

Informática Teórica y Aplicada

Por aquello de «aplicar principios y teorías» es que una disciplina científica se considera «aplicada», mientras que la disciplina que desarrolla las teorías se considera «teórica». A partir de estos dos conceptos de ciencia aplicada y ciencia teórica, se conciben, en las Ciencias Informáticas, estas dos ramas:

1. Informática teórica (ciencia formal o abstracta).
2. Informática aplicada (ciencia fáctica o empírica).

La **Informática Teórica** es la rama de las Ciencias Informáticas encargada de establecer los principios y teorías por los que se rigen las Ciencias Informáticas.

La **Informática Aplicada** es el resultado de aplicar los principios y teorías de la Informática Teórica, en la práctica.

El desarrollo de un lenguaje de programación forma parte de la Informática Aplicada.

Evolución de un lenguaje

En el contexto de un lenguaje de programación, se considera evolución al **lanzamiento de una versión mejorada del mismo lenguaje**.

Esta afirmación parte de la premisa de que —en el contexto teórico de las ciencias formales—, toda evolución implica un progreso, y que —en el mismo contexto—, todo progreso implica adelanto y mejora.

Principio de retrocompatibilidad

Uno de los principios en los que un lenguaje de programación se sustenta, es el «Principio de Retrocompatibilidad».

Este principio aplicado al intérprete de un lenguaje de programación, establece **«la capacidad que el intérprete de un lenguaje de programación, posee para interpretar programas desarrollados con una versión previa del mismo lenguaje»**.

Python y el principio de retrocompatibilidad

Un programa desarrollado en la versión 2.7 (versión inmediatamente anterior a la 3) no puede ser ejecutado sobre el intérprete de Python 3.0. Esto se debe principalmente, al reemplazo de la instrucción **print** por una función homónima, y a la eliminación de ciertas funciones como **raw_input** (entre otras). Esto supone una violación del principio de retrocompatibilidad.

Si bien pueden convivir intérpretes de diferentes versiones en un mismo sistema, solo una puede establecerse como predeterminada. Dado que los Sistemas Operativos dependen de algunos programas desarrollados en Python 2.7, actualizar la versión del intérprete de Python a la versión 3.0 o posterior, supondría la revisión de cientos de miles de líneas de código fuente.

Como respuesta a este supuesto, Python desarrolló una herramienta llamada **2to3** que permite migrar un paquete completo desde la versión 2 a la versión 3, suprimiendo así la necesidad de revisar y/o reescribir el código de un programa, manualmente. Esta herramienta se utiliza por línea de comandos, y se ejecuta mediante la siguiente instrucción:

```
2to3 -w carpeta/
```

Como dato adicional, mencionar que al no existir una herramienta 3to2 oficial (es decir, desarrollada y mantenida por Python), el proceso de conversión inversa no está soportado de forma nativa. No obstante, en un escenario favorable (en el que dicha herramienta existiese de forma oficial, y no a través de terceros), supondría el mantenimiento de 2 versiones de un mismo *script* o programa.

Decisión académica

Enseñar a programar en Python 2, implicaría formar un alumnado con conocimientos «obsoletos». Enseñar a programar en Python 3, implicaría no formar un alumnado con conocimientos suficientes para modificar un programa desarrollado en Python 2.

En términos académicos, elegir una u otra versión, supone un dilema no solo ético, sino también técnico. Por ese motivo, a lo largo de todos los cursos de la serie **«Python desde Cero»** se enseña al alumnado a **«programar en Python»**, y no así en una versión específica del lenguaje. De esta forma se pretende que el código fuente que el alumnado aprenda a desarrollar, pueda ser ejecutado indistintamente, tanto sobre el intérprete de Python 2, como sobre el intérprete de Python 3.

Para lograr esto, no son abarcadas funciones que no están disponibles en ambas ramas (2 y 3) del lenguaje, y a fin de solventar la incompatibilidad con la instrucción *print* y la función *raw_input* (indispensables en el *scripting*), a continuación, se provee al alumnado de un **hack de retrocompatibilidad** desarrollado específicamente para el estudiantado de los cursos «Python desde Cero» de la **Escuela de Informática Eugenia Bahit**. Dicho *hack* se expone en la página siguiente.

HACK DE RETROCOMPATIBILIDAD

Este *hack* consiste en la creación de dos funciones alternativas para imprimir y solicitar datos al usuario, respectivamente. Se denominan **echo()** y **get()**, por su familiaridad con los lenguajes GNU Bash y C.

```
from sys import version_info as version


def echo(data):
    if version.major == 2: exec("print \"{}\"".format(data))
    else: exec("print(\"{}\")".format(data))


def get(label):
    if version.major == 2: exec("data = raw_input(\"{}\")".format(label))
    else: exec("data = input(\"{}\")".format(label))
    return locals()['data']
```

Forma de uso

En el desarrollo de un *script*, las líneas precedentes serán escritas al comienzo del archivo.

Cuando se necesite imprimir se utilizará la función **`echo("cadena")`** en lugar de la instrucción *print* de Python 2 o la función *print()* de Python 3, mientras que para solicitar datos al usuario se empleará la función **`get("etiqueta")`**, en lugar de las funciones i*nput()* y *raw_input()* de las versiones 3 y 2 respectivamente.

(ver ejemplos de uso en la página siguiente)

Ejemplos de uso

Imprimir un texto en pantalla:

```
echo("Hola Mundo!")
# Equivalencia en Python 2: print "Hola Mundo!"
# Equivalencia en Python 3: print("Hola Mundo!")
```

Solicitar datos al usuario:

```
nombre = get("Nombre: ")
# Equivalencia en Python 2: nombre = raw_input("Nombre: ")
# Equivalencia en Python 3: nombre = input("Nombre: ")
```

BIBLIOGRAFÍA COMPLEMENTARIA

Documentación oficial del lenguaje Python: https://docs.python.org

CERTIFÍCATE

Demuestra cuánto has aprendido!

Si llegaste al final del curso puedes obtener una triple certificación:

- Certificado de **asistencia** (emitido por la escuela **Eugenia Bahit**)
- Certificado de **aprovechamiento** (emitido por **CLA Linux**)
- Certificación **aprobación** (emitido por **LAECI**)

Informáte con tu docente o visita la Web de certificaciones en http://python.laeci.org.

Si necesitas preparar tu examen, puedes inscribirte en el
Curso de Introducción al Lenguaje Python en la
Escuela de Informática Eugenia Bahit
www.eugeniabahit.com