

Introducción a la programación con Python



Agenda

1. Introducción
2. Elementos del lenguaje
3. Estructuras de Control
4. Funciones definidas por el usuario
5. Manejo de fechas
6. Manejo de Objetos
7. Paquetes y Módulos
8. Manejo de bases de datos

Introducción - ¿Qué es Python?

- Es un lenguaje de programación de alto nivel
 - **interpretado,**
 - **multiplataforma (Windows, Linux, OSX, Etc.),**
 - **tipado dinámico**
 - **multiparadigma (Modular, orientado a objetos y funcional):**

A diferencia de otros lenguajes de programación que no definen una postura en los estándares de codificación, Python define una serie de reglas de codificación

<https://www.python.org/dev/peps/pep-0008/#code-lay-out>

¿Por qué usar Python?

- **Multitud de herramientas en la librería estándar -> batteries included**
- **Campos muy variados** -> cómputo científico de alto rendimiento, con paquetes como Numpy, manejo de centros de datos con miles de servidores a través de proyectos como OpenStack, automatización industrial o la gestión de redes y un largo etc.
- **cantidad enorme de paquetes de terceros** -> disponibles a través del índice de paquetes de Python (PyPI)
- Tiene un excelente soporte en **desarrollo web** -> una oferta muy amplia de herramientas y frameworks, siendo Django y Flask de los más utilizados.
- El soporte de distintos sistemas de **gestión de bases de datos, SQL o NoSQL**, también caracteriza el ecosistema Python
- La disponibilidad de paquetes, sumado a la corta curva de aprendizaje, reducen dramáticamente el tiempo de desarrollo.

Filosofía

- Bello es mejor que feo.
- (se usan operadores lógicos and, or, en lugar de &&, | |).
- Explícito es mejor que implícito.
- (siempre se debe especificar el módulo al cual pertenece una función)
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.

Modalidades

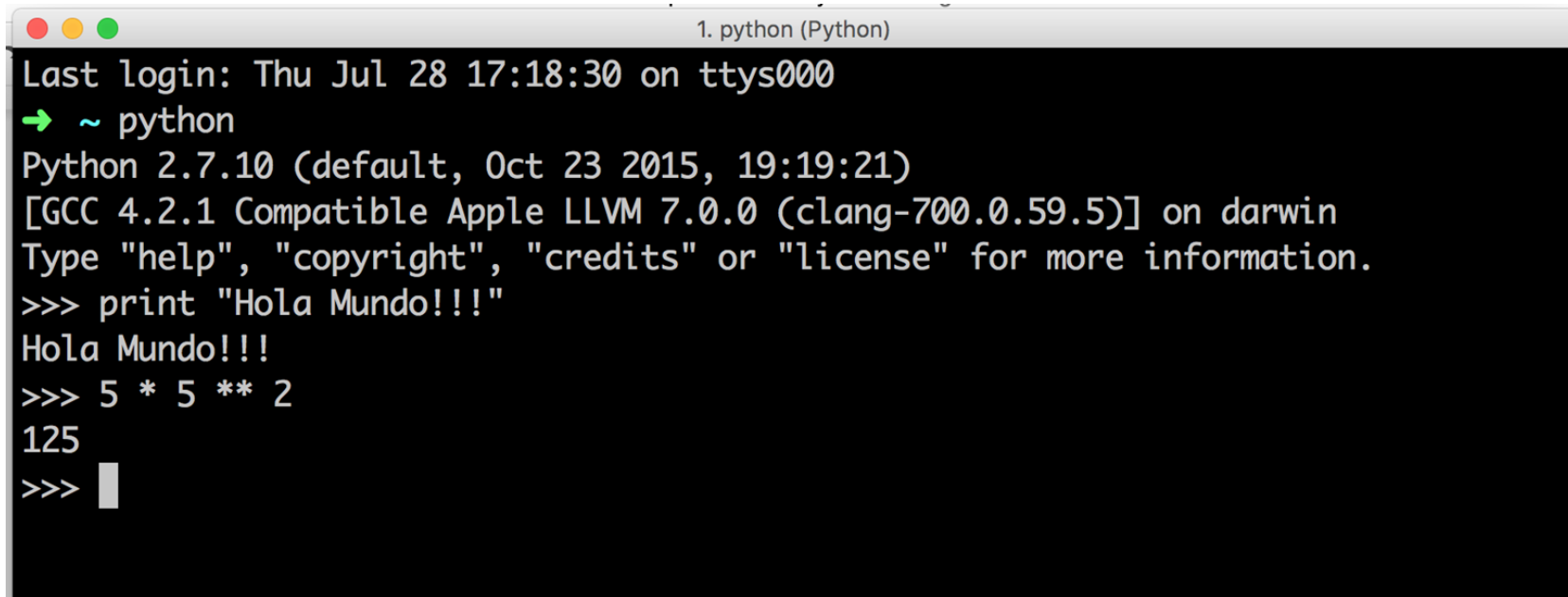
Python cuenta con dos modalidades para ejecutar el código.

- a) Modo interactivo o intérprete.

- a) Modo Script.

Modo Interactivo

Esta modalidad resulta bastante útil para probar fragmentos pequeños de código, funciones, operadores o expresiones. El intérprete evalúa la sentencia y regresa el resultado.

A screenshot of a terminal window titled "1. python (Python)". The window has a dark background with light-colored text. The terminal shows the following sequence of events: a login message "Last login: Thu Jul 28 17:18:30 on ttys000", a green arrow prompt followed by "~ python", the Python version and compiler information "Python 2.7.10 (default, Oct 23 2015, 19:19:21) [GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.59.5)] on darwin", a prompt to type help, copyright, credits, or license, and then two code snippets being executed: ">>> print 'Hola Mundo!!'" followed by the output "Hola Mundo!!", and ">>> 5 * 5 ** 2" followed by the output "125". The prompt ">>>" is followed by a white cursor bar.

```
1. python (Python)
Last login: Thu Jul 28 17:18:30 on ttys000
➔ ~ python
Python 2.7.10 (default, Oct 23 2015, 19:19:21)
[GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.59.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hola Mundo!!!"
Hola Mundo!!!
>>> 5 * 5 ** 2
125
>>> █
```

b) Modo Script.

- Esta modalidad se basa en la idea de un conjunto de sentencias almacenadas en un archivo con extensión **.py** (Script).
- El script completo es interpretado por la máquina virtual de Python.
- Ejercicio: HolaMundo.py
 - Elaborar un programa que pida el nombre de la persona a saludar y escriba el siguiente mensaje "Hola NOMBRE, ¿Como estas?"

Ejemplo hola_mundo.py

Python 2.X

```
# -*- coding: utf-8 -*-
nombre= raw_input("Introduce tu nombre:")
print "Hola", nombre, "¿qué tal?"
raw_input("pulse tecla para terminar")
```

Python 3.X

```
# -*- coding: utf-8 -*-
nombre= input("Introduce tu nombre:")
print ("Hola", nombre, "¿qué tal?")
input("pulse tecla para terminar")
```

Funciones de entrada / salida

Salida:

print.- Se utiliza para emitir una salida, por default sobre la pantalla. La sentencia recibe un array de objetos a imprimir separados por comas.

```
print 'hola', 5, ' ', '\n', 'Fin'
```

Es posible crear plantillas de cadenas método format de objeto string.

Sintaxis:

```
print 'Hola {}, tu edad es de: {}'.format('Emir', 34)
```

En Python 3 print es una funcion necesita parentesis y tiene otros parametros)

Function de entrada

input / raw_input: Permiten leer datos desde el teclado.
sintaxis: `input(cadena_mensaje)`

Ejemplo:

```
nombre = raw_input('Ingresa el nombre:')  
edad = input('Ingresa la edad:')
```

`raw_input` -> esta deprecada en la versión de Python 3.0., en la version 2 devuelve como String el texto introducido por el usuario

`Input` -> devuelve el valor con el tipo de lo introducido por el usuario en python 2. En python 3 sustituye a `raw_input` y se utiliza igual que `raw_input` en Python2.(siempre string)

Más diferencias significantes entre Python 2 y 3:

<https://www.pythonmania.net/es/2016/02/29/las-principales-diferencias-entre-python-2-y-3-con-ejemplos/>

Elementos del lenguaje

Variables

Un espacio en memoria para almacenar datos. La sintaxis en Python para definir una variable es:

variable = valor

El valor que se le asigna a la variable define su tipo de dato, y las operaciones que se pueden aplicar sobre dicha variable.



Convenciones de Python para nombrar variables:

1. Utilizar nombres descriptivos.
2. Usar letras minúsculas.
3. En caso de ser un nombre compuesto, separar por guión bajo.


Ejemplo: `folio_fiscal = 10`

1. Antes y después del signo = debe haber un solo caracter blanco.

Ejemplo de asignación de variables

```
mysql (mysql)  %1  ~ (zsh)  %2  python (Python)  %3  
>>> a = 1  
>>> b, c = 1, "hola"  
>>> d = e = f = 4  
>>> print "a = %d b = %d c = %s d = %d e = %d f = %d" % (a, b, c, d, e, f)  
a = 1 b = 1 c = hola d = 4 e = 4 f = 4  
>>> █
```

Constantes

 Python NO define un mecanismo especial para definir constantes.

Si se utiliza una variable como constante, el nombre debe estar en mayúsculas, ejemplo:

```
TASA_IVA = 16.00
```


Tipo de datos primitivos

Numéricos:

Enteros

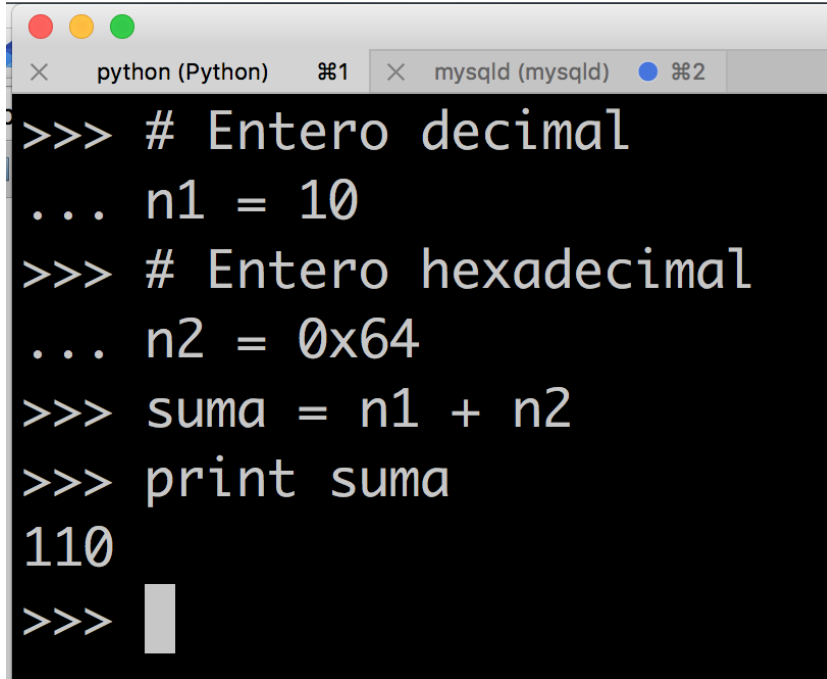
Flotantes

Complejos

Cadenas

Booleanos

Tipo de dato numérico entero



```
>>> # Entero decimal
... n1 = 10
>>> # Entero hexadecimal
... n2 = 0x64
>>> suma = n1 + n2
>>> print suma
110
>>> 
```

Tipo de dato numérico real.

```
>>> n1 = 0x64
>>> n2 = 10.5
>>> suma = n1 + n2
>>> print suma
110.5
```

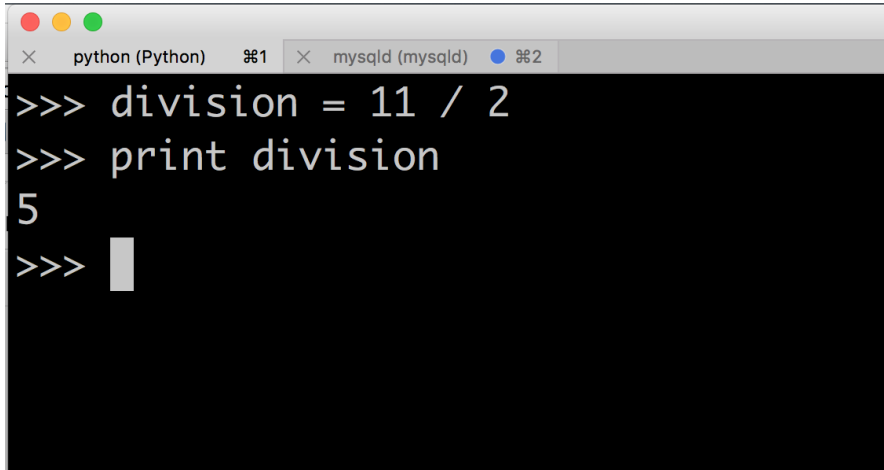
Tipo de datos booleano

```
>>> verdad = True
>>> mentira = False;
>>>
```

Operadores aritméticos

Operador	Operación
+	Suma
-	Resta
*	Multiplicación
**	Exponenciación
/	División
//	División Entera
%	Residuo
-	Negación.

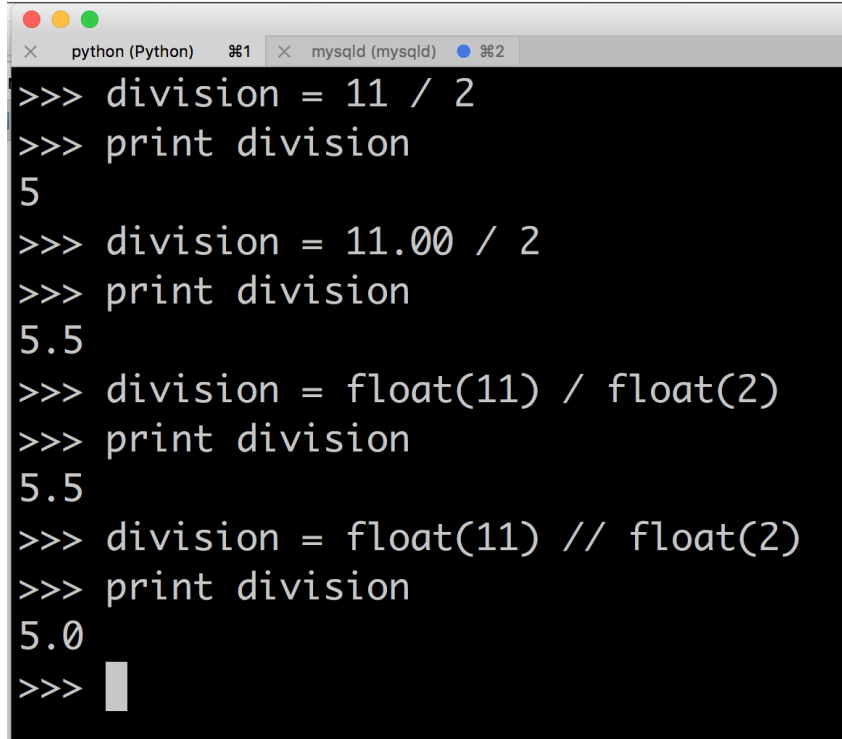
```
python (Python) 361  mysql (mysql) 362
>>> suma = 10 + 15
>>> print 'Suma =', suma
Suma = 25
>>> resta = 15 - 10
>>> print 'Resta = ', resta
Resta = 5
>>>
>>> multiplicacion = 10 * 2
>>> print 'Multiplicación = ', multiplicacion
Multiplicación = 20
>>>
>>> division = 10 / 2
>>> print 'División = ', division
División = 5
>>>
>>> exponenciacion = 2 ** 3
>>> print 'Exponenciacion = ', exponenciacion
Exponenciacion = 8
>>>
>>> negacion = - exponenciacion
>>> print negacion
-8
```



A screenshot of a Python terminal window. The window has a title bar with three colored buttons (red, yellow, green) and two tabs: 'python (Python)' and 'mysqld (mysqld)'. The terminal content shows the following interaction:

```
>>> division = 11 / 2
>>> print division
5
>>> █
```

¿Correcto?



```
>>> division = 11 / 2
>>> print division
5
>>> division = 11.00 / 2
>>> print division
5.5
>>> division = float(11) / float(2)
>>> print division
5.5
>>> division = float(11) // float(2)
>>> print division
5.0
>>> 
```

*Ojo esto cambia en Python 3

Orden de precedencia

1. Los paréntesis se usan para agrupar expresiones . ()
2. Exponenciación **
3. Operadores unarios -, +
4. Multiplicar, dividir, módulo y división entera. *, /, //, %
5. Suma, Resta +, -

Ejercicio:

Elaborar un programa en Python que lea desde el teclado el subtotal de la venta, % de descuento y tasa de IVA. Calcular el total, ejemplo:

```
subtotal: 100
% descuento: 50
Tasa de IVA: 16.00
=====
Total: 58.00
Descuento: 50.00
IVA: 8.00
```

Guideline sobre operadores aritméticos:



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

Comentarios

Los scripts Python pueden incluir comentarios (notas que como programadores, indicamos en el código para poder comprenderlo mejor).

Los comentarios pueden ser de dos tipos:

De una sola línea

Multilínea.

Comentario de una línea

```
>>> # leer datos
... subtotal = input('Ingrese el subtotal:')
Ingrese el subtotal:10
>>> # Calcular el iva
... iva = subtotal * 0.16
>>> # imprimir
... print subtotal, iva
10 1.6
>>> █
```

Comentario multilínea



A screenshot of a Python terminal window. The window has a title bar with three colored buttons (red, yellow, green) on the left and the text "1. python (Python)" on the right. Below the title bar is a tab bar with three tabs: "python (Python) %1", "mysqld (mysqld) %2", and "python (Python) %3". The main area of the window is black with white text. The text shows a multi-line comment in Python, starting with three greater-than signs followed by three quotes, then the text "comentario multi linea" on the first line, "... para ejemplificar" on the second line, and a third line starting with three dots followed by a long string of equals signs and three quotes. The comment is closed on the fourth line with a single quote, followed by "comentario multi linea\npara ejemplificar \n" and another long string of equals signs, and finally a single quote. The prompt ">>>" appears at the end of the fourth line.

```
>>> """ comentario multi linea
... para ejemplificar
... ===== """
' comentario multi linea\npara ejemplificar \n===== '
>>>
```

Guidelines sobre comentarios



Los comentarios en línea deben separarse de la sentencia con dos espacios en blanco. Luego del símbolo **#** debe ir un solo espacio.

Correcto

```
a = 15  # edad de María
```

Incorrecto

```
a = 15 # edad de María
```

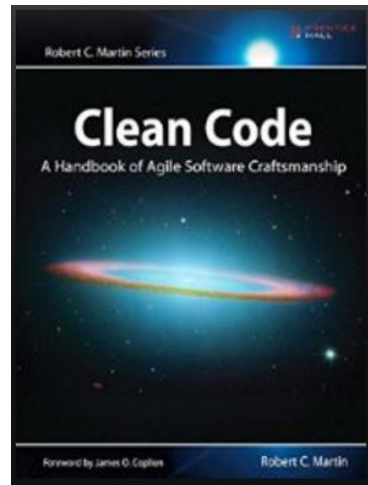
Más sobre comentarios (Clean code)

No comentes mal código, reescribelo.

Los comentarios no pueden maquillar el mal código.

La necesidad de comentarios para aclarar algo es síntoma de que hay código mal escrito que debería ser rediseñado.

Es preferible expresarse mediante el propio código.



Situaciones donde los comentarios son útiles:

Aspectos legales del código.

Advertir de consecuencias.

Comentarios TODO.

Remarcar la importancia de algo.

Javadocs en APIs públicas.

Ejemplo:

```
1 # Solicitar datos
2 subtotal = input('Ingrese el subtotal: ')
3 porc_descuento = input('Ingrese el descuento: ')
4 tasa_iva = input('Ingrese la tasa de I.V.A: ')
5
6 # Calcular datos
7 descuento = subtotal * (porc_descuento / 100.00)
8 subtotal_menos_descuento = subtotal - descuento
9 importe_iva = subtotal_menos_descuento * (tasa_iva / 100.00)
10 total = subtotal_menos_descuento + importe_iva
11
12 # Imprimir resultado
13 print "=====
14 print 'Total: ', total
15 print 'Descuento: ', descuento
16 print 'IVA:', importe_iva
```

Sustituir comentarios por funciones auto descriptivas.

```
def solicitar_datos():
    subtotal = input('Ingrese el subtotal: ')
    porc_descuento = input('Ingrese el descuento: ')
    tasa_iva = input('Ingrese la tasa de I.V.A: ')

def calcular_datos():
    descuento = subtotal * (porc_descuento / 100.00)
    subtotal_menos_descuento = subtotal - descuento
    importe_iva = subtotal_menos_descuento * (tasa_iva / 100.00)
    total = subtotal_menos_descuento + importe_iva

def imprimir_resultado():
    print "=====
    print 'Total: ', total
    print 'Descuento: ', descuento
    print 'IVA:', importe_iva

solicitar_datos()
calcular_datos()
imprimir_resultado()
```

Tipo de datos complejos:

- Python posee 3 tipos de datos más, que permiten almacenar 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 **un objeto que permite almacenar varios datos inmutables** (no pueden ser modificados una vez creados) de tipos diferentes.

La colección de datos es heterogénea.

Se pueden construir con la función **tuple**(colección)

Uso de tuplas

```
Python 2.7.12 (v2.7.12:d33e0cf91556, Jun 27 2016, 15:19:22) [MSC v.1500 32 bit  
Intel)] on win32
```

```
Type "copyright", "credits" or "license()" for more information.
```

```
>>> tupla=()
```

```
>>> print "Tupla:", tupla
```

```
Tupla: ()
```

```
>>> mi_tupla=(1,'Victor','Custodio', 30)
```

```
>>> print "Tupla con valores" , mi_tupla
```

```
Tupla con valores (1, 'Victor', 'Custodio', 30)
```

```
>>> #intento modificar mi_tupla:
```

```
>>> mi_tupla[0]=6
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#5>", line 1, in <module>
```

```
    mi_tupla[0]=6
```

```
TypeError: 'tuple' object does not support item assignment
```

```
>>> mi_tupla[4]="nuevo valor"
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#6>", line 1, in <module>
```

```
    mi_tupla[4]="nuevo valor"
```

```
TypeError: 'tuple' object does not support item assignment
```

```
>>> print "Mi tupla",mi_tupla
```

```
Mi tupla (1, 'Victor', 'Custodio', 30)
```

```
>>> |
```

Más sobre tuplas

Se puede acceder a cada uno de los datos mediante su índice correspondiente, siendo 0 (cero), el índice del primer elemento:

```
print 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:

```
print mi_tupla[1:4] # Devuelve: (15, 2.8, 'otro dato')
print mi_tupla[3:]  # Devuelve: ('otro dato', 25)
print 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:

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

Listas

Una lista es similar a una tupla, con la diferencia fundamental de que permite modificar los datos una vez creados:

```
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:

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

Las listas, NO son inmutables: permiten modificar los datos una vez creados:

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

Las listas, a diferencia de las tuplas, permiten agregar nuevos valores:

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

Métodos de las listas

Método	Descripción
append(x)	Agrega x al final de la lista.
insert(i, x)	Inserta x en el índice de la lista.
remove(x)	Elimina la primera ocurrencia de x en la lista
sort(key=comparer)	Ordena los elementos de la lista.
reverse()	Genera una lista invertida
pop(i)	Elimina el elemento i de la lista y lo regresa como resultado.
extend(L)	Añade todos los elementos de L a la lista
count(x)	Devuelve la cantidad de veces que X aparece en la lista.
index(x)	Devuelve el índice de X de la lista

Ejemplo de ordenamiento de listas sort

```
>>> l = ["1", "10", "2", "1221", "30"]
>>> def comparar(x):
...     return int(x)
...
>>> l.sort()
>>> l
['1', '10', '1221', '2', '30']
>>> l.sort(key = comparar)
>>> l
['1', '2', '10', '30', '1221']
>>>
```

Diccionarios

Mientras que a las listas y tuplas se accede solo y únicamente por un índice numérico, los diccionarios permiten utilizar una clave para declarar y acceder a un valor:

```
mi_diccionario = {'clave_1': 'valor_1', 'c_2': 'valor_2', 'clave_7': 'valor_7'}  
print mi_diccionario['clave_2'] # Salida: valor_2
```

Un diccionario permite eliminar cualquier entrada:

```
del(mi_diccionario['c_2'])
```

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

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

Métodos del objeto diccionario

Método	Descripción
has_key(k)	Regresa True si el diccionario contiene la llave especificada en k.
items()	Devuelve una lista con los elementos del diccionario. La lista esta compuesto por tuplas de (llave, valor)
keys()	Devuelve una lista con las llaves del diccionario.
values()	Devuelve una lista con los valores del diccionario.
get(k)	Regresa el valor del elemento que contiene la llave k.
get(k,x)	Regresa el valor del elemento que contiene la llave k. en caso de no encontrar regresa el valor X.
clear()	Limpia el diccionario.
popitem()	Remueve un elemento al azar.
update(dic2)	Actualiza o inserta los elementos de dic2.

Estructuras de Control

Estructura de control condicionales

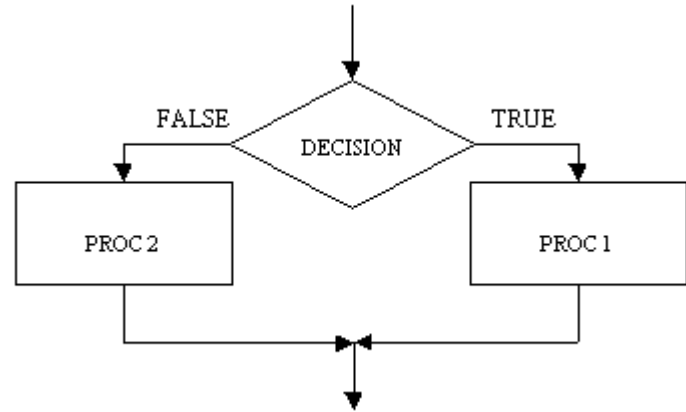
Nos permiten evaluar condiciones y la acción que vamos a ejecutar si esta se cumple o no.

La evaluación de condiciones, sólo **puede arrojar verdadero o falso** (True o False). Para describir la evaluación a realizar sobre una condición, se utilizan **operadores relacionales**:

Símbolo	Descripción	Ejemplo	Resultado
==	Igual que	5 == 7	False
!=	Diferente	'rojo' != 'Rojo'	True
<	Menor que	8 < 12	True
>	Mayor que	12 > 7	True
<=	Menor igual	12 <= 12	True
>=	Mayor Igual	4 >= 5	False

Sentencia IF

```
if condition_1:  
    statement_block_1  
elif condition_2:  
    statement_block_2  
else:  
    statement_block_3
```



Ejercicio: Algoritmo de Edad Canina.

Es una creencia generalmente aceptada, asumir que un año en la vida de un perro corresponde a **7** años en la vida del ser humano. Otro método es el siguiente:

Un perro de **1** año de edad corresponde aproximadamente a un niño de **14** años edad.

Un perro que es de **2** años de edad corresponde a una edad de **22** años humana.

Cada año adicional del perro corresponde a **5** años humanos.

Elabore un programa en Python que implementa dicho algoritmo.

Falso o verdadero

Las siguientes expresiones se interpretan como **Falso** en Python:

Números con valor cero (0, 0L, 0.0, 0x00).

El literal **False**.

Cadenas vacías.

Listas, Diccionarios o tuplas vacías.

El valor especial **None**.

Cualquier otro valor se considera como verdadero (**True**)º

Abreviaciones del IF

Programadores de C utilizan la siguiente notación para abreviar un bloque *IF*:

```
max = (a > b) ? a : b;
```

Esta abreviación representa el siguiente código de IF:

```
if (a > b)
    max = a;
else
    max = b;
```

La notación en Python equivalente sería:

```
max = a if (a > b) else b;
```

Abreviación del IF

```
1 a = input('Ingrese el primer numero: ')\n2 b = input('Ingrese el segundo numero: ')\n3 mayor = a if a > b else b;\n4 print 'El Mayor de %s y %s, es: %s' % (a, b, mayor)
```

Ciclos

Los ciclos permiten ejecutar un bloque de instrucciones “n” veces, mientras una condición determinada se este cumpliendo.

Python provee 2 tipos de ciclos:

while

for

Sintaxis Ciclo **While**

```
while <condicion>:  
    sentencia1  
[else:  
    sentencia2  
]
```

Ejemplo de ciclo While.

```
>>> anio = 2001
>>> while anio <= 2012:
...     print "Informes del año %s" % (anio)
...     anio += 1
...
Informes del año 2001
Informes del año 2002
Informes del año 2003
Informes del año 2004
Informes del año 2005
Informes del año 2006
Informes del año 2007
Informes del año 2008
Informes del año 2009
Informes del año 2010
Informes del año 2011
Informes del año 2012
```

Ejercicio: Adivina el número

Escribe un programa Python para implementar el juego “Adivina el número”

1. El programa solicita el nombre del jugador.
2. El programa obtiene un número aleatorio entre 0 y 100.
3. El programa le pregunta al jugador el número que adivino.
4. El jugador ingresa el número
5. Si el número indicado por el jugador es menor, el programa desplegará el siguiente mensaje:
 - a. “Sorry {nombre}, tu elección de {numero} fue muy BAJA.”
6. Si el número indicado por el jugador es mayor, el programa desplegará el mensaje :
 - a. “Sorry {nombre}, tu elección de {numero} fue muy Alta.”
7. El programa termina hasta que el jugador adivine el número
 - a. El programa desplegará el mensaje:
 - i. Excelente trabajo {nombre}, Tú ganas, era el {numero_aleatorio}!

Tip:

El Paquete **random** tiene diversas funciones para obtener números aleatorios.

ejemplo:

```
print random.randint(0,100)    # obtiene un entero aleatorio entre 0 y 100
```

Ejemplo para obtener números aleatorios

```
>>> import random
>>> print random.randint(0, 100)
97
>>> print random.randint(0, 100)
39
>>> print random.randint(0, 100)
82
>>> print random.randint(50, 100)
91
>>>
```

Adivina el número con validación de nombre.

Adivina el número con validación de nombre y contabilidad de intentos.

Ciclo For

Se utiliza para iterar sobre una estructura de datos compleja (lista o tupla),

Sintaxis:

```
for nombre_iterador in objeto_coleccion:  
    sentencia  
    sentencia2  
    ...  
    sentenciaN
```

Es posible abandonar el cuerpo de un ciclo utilizando la sentencia **break**,

Es posible avanzar a la siguiente iteración con la sentencia **continue**.

Ejemplo con listas:

```
>>> lista = ['Lunes', 'Martes', 'Miercoles', 'Jueves', 'Viernes', 'Sabado', 'Domingo']
>>> for iterador in lista:
...     print iterador
...
Lunes
Martes
Miercoles
Jueves
Viernes
Sabado
Domingo
>>> █
```

Ejemplo con tuplas

```
>>> color = ('Rojo', 'Verde', 'Azul', 'Amarillo')
>>> for iterador in color:
...     print iterador
...
Rojo
Verde
Azul
Amarillo
>>> █
```

Función range(n)

Genera una lista con los números desde 0 hasta n-1.

Ejemplo:

```
>>> lista = range(10)
>>> for n in lista:
...     print n
...
0
1
2
3
4
5
6
7
8
9
>>> █
```

Puede recibir 3 parametros:
Range(inicio=0,fin,salto=1)

FUNCIONES DEFINIDAS POR EL USUARIO

Funciones definidas por el usuario

En Python las funciones se declaran mediante la palabra reservada **def**.

La sintaxis básica es:

```
def nombre_funcion():  
    #sentencias
```

Una función no es ejecutada hasta que es invocada, para invocar la solo hay que poner el nombre de la función con la apertura y cierre de paréntesis, ejemplo:

```
nombre_funcion()
```

Funciones

Para que una función regrese un valor a la expresión que la invoca se debe utilizar la expresión **return**.

ejemplo:

```
def hola_mundo(nombre):  
    return "Hola {}, ¿Que tal?".format(nombre)  
print hola_mundo("Víctor")
```

Los argumentos de la función se indican como una lista de variables, separados por comas.

Es posible definir argumentos opcionales, asignándoles un valor por default.

```
def nombre_completo(nombre, apellido_paterno, separador = ' '):  
    return nombre + separador + apellido_paterno  
nombre_completo('Víctor', 'Custodio')
```


Lista de Parámetros

En caso de querer definir una lista de parámetros indeterminada se utiliza la siguiente notación:

```
def sumatoria(*numeros):  
    resultado = 0  
    for n in numeros:  
        resultado += n  
    return resultado  
print sumatoria(1,5,6,5)
```

La función `len(numeros)` devuelve el número de parámetros.

Diccionario de Parámetros

En caso de querer definir un diccionario de parámetros indeterminada se utiliza la siguiente notación:

```
def sumatoria(**kwargs):  
    for key, value in kwargs.items():  
        print "{} = {}".format(key, value)
```

Manejo de Cadenas paquete string

Constantes predefinidas en el paquete string.

Constante	Valor
ascii_letters, letters	"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
ascii_lowercase, lowercase	"abcdefghijklmnopqrstuvwxyz"
ascii_uppercase, uppercase	"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
digits	'0123456789'
hexdigits	'0123456789abcdefABCDEF'
punctuation	'!"#\$%&\'()*+,-./:;<=>?@[\\]^_`{ }~'
printable	'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#\$%&\'()*+,-./:;<=>?@[\\]^_`{ }~ \t\n\r\x0b\x0c'
whitespace	'\t\n\r\x0b\x0c\r '

Strings

Strings son un tipo de dato muy poderoso en Python.

Para crearlas solo hay que encerrar una cadena entre comillas dobles o simples.

```
cadena = "Hello World!"
```

```
cadena2 = "Pythong programming"
```

Python no soporta el tipo de dato Char. Estos son tratados como cadenas de longitud 1.

Para acceder a un substring se utiliza []

```
>>> cadena1 = "Hello World!"
>>> cadena2 = "Pythong programming"
>>> print cadena1[0]
H
>>> print cadena2[2:5]
tho
>>> print cadena2[1:5]
ytho
>>> 
```

Caracteres de escape

Notación diagonal invertida	Descripción
\a	Campana
\b	Backspace
\e	Escape
\f	Avance de página
\n	Nueva línea
\r	Retorno de carro
\s	Espacio
\t	Tab
\v	Vertical <u>tab</u>
\x	Carácter x
\xnn	Carácter hexadecimal

Operadores especiales de Strings

Operador	Descripción
+	Concatenación
*	Repetición (Concatenación múltiple)
[]	Slice
[:]	Range Slice – Carácter de un rango particular
in	Es miembro – Regresa True si el carácter existe en la cadena
%	Formato

```
>>> print "Emir " + "Abel"
Emir Abel
>>> print "=" * 40
=====
>>> print "Emir"[0]
E
>>> print "Emir"[0:2]
Em
>>> print "E" in "Emir"
True
>>> print "X" in "Emir"
False
>>> print "Hola %s " % ( "Emir")
Hola Emir
```

Balanceo de paréntesis

Elaborar una programa Python que lea una cadena del teclado y determine si el número de paréntesis ingresados esta balanceado.

Los paréntesis que se pueden usar son los siguientes:

(), { }, []

Ejemplo:

(() -> No válido (5+5) → valido

((5+2*4)] → No válido

Funciones Matemáticas

Paquete math

Funciones Matemáticas

Metodos de modulo Math

```
* floor(x).- El entero mas grande no mayor que x. math.floor(5.9) => 5.0
* ceil(x) .- El entero mas pequeño mayor que igual x. math.ceil(5.5) => 6.0
* log(x) .- El logaritmo natural de x. X mayor a cero math.log(11) => 2.3978952728
* log10(x).- El logaritmo base 10 de x. X mayor a cero math.log10(14) => 1.14612803568
* modf(x) .- Obtiene una tupla con los componentes de un numero real. math.modf(5.5) => (0.5, 5.0)
* pow(x,y) .- Obtiene el valor de x ** y. math.pow(2, 3) => 8.0
* sqrt(x) .- Obtiene la raiz cuadrada de x. math.sqrt(16) => 4.0
```

Métodos predefinidos

```
* abs(x).-Valor absoluto de x. abs(-5) => 5
* max(*n).-El mayor elemento de la lista max(1,4,5,6,8, 12) => 12
* min(*n).-El menor elemento de la lista min(1,4,5,6,8, 12) => 1
* round(x,[y]) .- Redondea el numero x a y decimales round(0.5) => 1.0
* round(4.523, 2) => 4.52
```

Código fuente: [mathEjemplo.py](#)

Código fuente:

```
1  # -*- encoding: utf-8 -*-
2  import math
3  print 'Metodos de modulo Math'
4
5  print "* floor(x).- El entero mas grande no mayor que x.  math.floor(5.9) => ", math.floor(5.9)
6  print "* ceil(x) .- El entero mas pequeño mayor que igual x.  math.ceil(5.5) => ", math.ceil(5.5)
7  print "* log(x) .- El logaritmo natural de x. X mayor a cero  math.log(11) => ", math.log(11)
8  print "* log10(x).- El logaritmo base 10 de x. X mayor a cero  math.log10(14) => ", math.log10(14)
9  print "* modf(x) .- Obtiene una tupla con los componentes de un numero real. math.modf(5.5) => ", math.modf(5.5)
10 print "* pow(x,y) .- Obtiene el valor de x ** y. math.pow(2, 3) => ", math.pow(2, 3)
11 print "* sqrt(x) .- Obtiene la raiz cuadrada de x. math.sqrt(16) => ", math.sqrt(16)
12
13 print
14 print 'Métodos predefinidos'
15 print "* abs(x).-Valor absoluto de x.  abs(-5) => ", abs(-5)
16 print "* max(*n).-El mayor elemento de la lista  max(1,4,5,6,8, 12) => ", max(1,4,5,6,8, 12)
17 print "* min(*n).-El menor elemento de la lista  min(1,4,5,6,8, 12) => ", min(1,4,5,6,8, 12)
18 print "* round(x,[y]) .- Redondea el numero x a y decimales round(0.5) => ", round(0.5)
19 print "*
                                round(4.523, 2) => ", round(4.523, 2)
```

Funciones de trigonometría

Metodos de modulo Math

```
* acos(x).- Retorna el arcocoseno de x. math.acos(0.5)==> 1.0471975512
* asin(x).- Retorna el arcoseno de x. math.asin(0.5)==> 0.523598775598
* atan(x).- Retorna el arcotangente de x. math.atan(0.5)==> 0.463647609001
* cos(x).- Retorna el coseno de x. math.cos(0.5)==> 0.87758256189
* sin(x).- Retorna el seno de x. math.sin(0.5)==> 0.479425538604
* tan(x).- Retorna el tangente de x. math.tan(0.5)==> 0.546302489844
* degrees(x).- Convierte el angulo x en radianes a grados. math.degrees(0.5)==> 28.647
8897565
* radians(x).- Convierte el angulo x en grados a radianes. math.radians(28.6478897565)
==> 0.499999999999
```

Constantes del módulo math

```
math.e ==> 2.71828182846
math.pi ==> 3.14159265359
```

Más sobre cadenas

Métodos del objeto string

`capitalize()`.- Pone el primer carácter en mayúsculas, y el resto en minúsculas.

Ejemplo:

```
print "HOY ES LUNES".capitalize()
```

`center(w)`.- Rellena con espacios a la izquierda y a derecha para centrar el texto. ejemplo:

```
print "Hoy es Lunes".center(40)
```

`count(str,begin=0, lon=len(str))`.- cuenta el número de veces que un substring aparece dentro de una cadena. Ejemplo

```
print "# -*- coding: utf-8 -*-".count("-*-") # 2
```

`find(str,beg=0,len=len(str)).` – Regresa la posición donde aparece el substring.

ejemplo:

```
print "# -*- coding: utf-8 -*-".find("-*-") # 2
```

`isalpha().` – regresa verdadero si todos los caracteres son letras. ejemplo:

```
print "Emir2".isalpha(), "emir".isalpha()
```

`isdigit().` – Regresa verdadero si todos los caracteres son dígitos (0-9)

```
print "5.5".isdigit(), "0012932343".isdigit()
```

`len(str).` – Regresa la longitud de la cadena.

```
print len("5.5"), len("0012932343")
```

`lower(str).` – Convierte una cadena a minúsculas. Ejemplo:

```
print "HOY ES LUNES".lower()
```

`upper(str).` – Convierte una cadena a mayúsculas. Ejemplo:

```
print "Hoy es lunes".upper()
```

`replace(old, new).`- Reemplaza una subcadena por otra. Ejemplo:

```
print "HOY ES LUNES".replace("LUNES", "Miercoles")
```

`split(str).`- Genera una lista a partir de una cadena Ejemplo:

```
print "HOY ES LUNES".split(' ') # ['HOY', 'ES', 'LUNES']
```

`zfill(width).`- Rellena una cadena con ceros a la izquierda, hasta completar la longitud de `width`. Ejemplo:

```
print "1234".zfill(5)
```


format(*args,**kwargs)

```
>>> cadena = "bienvenido a mi aplicación {0}"
```

```
>>> print cadena.format("en Python")
```

```
bienvenido a mi aplicación en Python
```

```
>>> cadena = "Importe bruto: ${0} + IVA: ${1} = Importe neto: {2}"
```

```
>>> print cadena.format(100, 21, 121)
```

```
Importe bruto: $100 + IVA: $21 = Importe neto: 121
```

```
>>> cadena = "Importe bruto: ${bruto} + IVA: ${iva} = Importe neto: {neto}"
```

```
>>> print cadena.format(bruto=100, iva=21, neto=121)
```

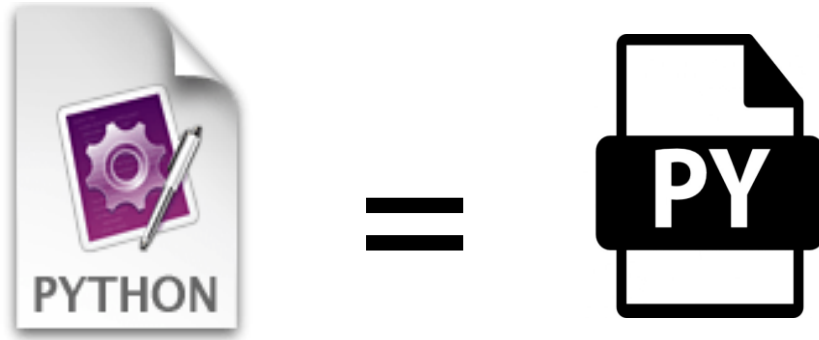
```
Importe bruto: $100 + IVA: $21 = Importe neto: 121
```

```
>>> print cadena.format(bruto=100, iva=100 * 21 / 100, neto=100 * 21 / 100 + 100)
```

```
Importe bruto: $100 + IVA: $21 = Importe neto: 121
```

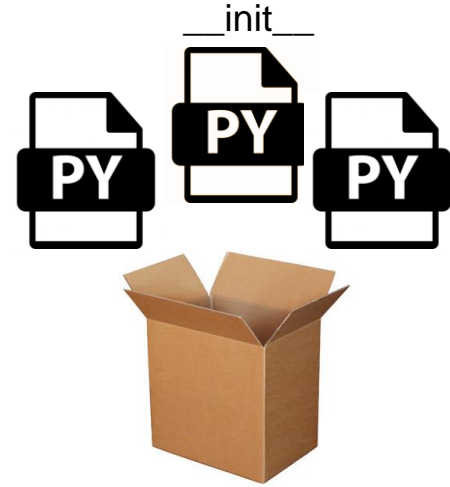
MÓDULOS Y PAQUETES

Módulos y Paquetes



Un archivo de python, es un archivo con extensión “.py” que a su vez es un módulo.

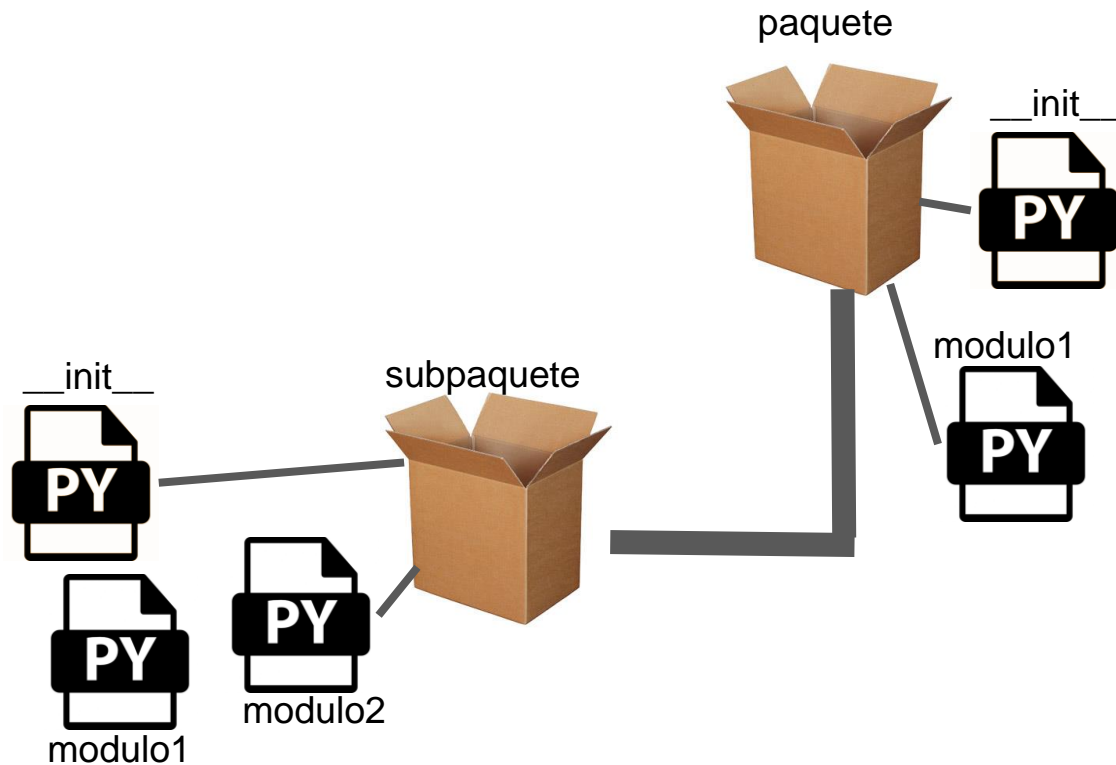
Módulos y Paquetes



Un paquete, es una carpeta que puede contener uno o más módulos, pero para ser considerado como paquete, debe contener un archivo de inicio llamado “`__init__.py`” aún cuando este archivo se encuentre vacío.

Módulos y Paquetes

Los paquetes a su vez
pueden contener
subpaquetes



Módulos y Paquetes

Los módulos no
necesariamente
deben estar dentro
de un paquete

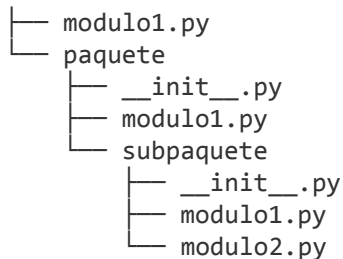
```
├── modulo1.py
└── paquete
    ├── __init__.py
    ├── modulo1.py
    └── subpaquete
        ├── __init__.py
        ├── modulo1.py
        └── modulo2.py
```

Módulos y Paquetes

Para usar el contenido de cada módulo desde otros módulos es necesario importar los módulos, para hacer dicha tarea, es necesario utilizar la palabra `import` seguida del nombre del paquete (de ser necesario) y el nombre del módulo sin su extensión.

Por ejemplo:

```
import modulo1;  
#importa un módulo que no está en paquete  
  
import paquete.modulo1;  
#importa un módulo que está en un paquete  
  
import paquete.subpaquete.modulo1;
```



Módulos y Paquetes

“Los namespaces”

Para acceder desde el módulo actual hacia cualquier elemento del módulo importado se utiliza un “namespace” (una ruta definida por la forma en que el módulo fue importado), seguido de un punto y el nombre del elemento que se desea obtener. Por ejemplo:

```
print modulo1.CONSTANTE_1;
```

```
print paquete.modulo1.CONSTANTE_1;
```

```
print paquete.subpaquete.modulo1.CONSTANTE_1;
```


Módulos y Paquetes

“Los alias”

Es una forma de acceder de una manera más corta a los módulos que importamos, por ello, al momento de la importación, utilizamos la palabra “as” para definir el alias.

Al importar

```
import modulo1 as m;
```

```
import paquete.modulo1 as pm;
```

```
import paquete.subpaquete.modulo1 as psm;
```

```
print m.CONSTANTE_1;
```

```
print pm.CONSTANTE_1;
```

```
print psm.CONSTANTE_1;
```

Módulos y Paquetes

¿Utilizar valores sin namespaces?

Si esa duda no te dejaba respirar tranquilamente, respira, es posible utilizar los valores sin namespaces, solo debes cambiar la forma en que se importa el módulo y podrás hacerlo sin contratiempos.

Para poder hacer uso de esta bondad del lenguaje solo necesitamos hacer uso de la instrucción “from” seguida de nuestro “namespace” más el nombre del módulo que deseamos importar

```
from paquete.modulo1 import CONSTANTE_1
```

```
print CONSTANTE_1
```

En este caso, se accederá directamente al elemento, sin recurrir a su namespace:

Módulos y Paquetes

¿Utilizar valores sin namespaces?

También es posible importar más de un elemento en la misma instrucción, para que seas capaz de lograr esto, solo es necesario que separes cada atributo por una coma y un espacio en blanco:

```
from paquete.modulo1 import CONSTANTE_1, CONSTANTE_2
```

Módulos y Paquetes

¿Utilizar valores sin namespaces?

Y... ¿Qué pasa si los elementos que necesito importar están en módulos diferentes pero tienen los mismos nombres?

Para evitar equivocarnos utilizamos alias en esos elementos.

```
from paquete.modulo1 import CONSTANTE_1 as C1, CONSTANTE_2 as C2
from paquete.subpaquete.modulo1 import CONSTANTE_1 as CS1, CONSTANTE_2 as CS2
```

```
print C1
print C2
print CS1
print CS2
```

Módulos y Paquetes

¿Utilizar valores sin namespaces?

Y... ¿Qué pasa si no quiero utilizar namespace pero tampoco alias?

Se puede hacer, pero es poco recomendable debido a que puedes confundirte. En este caso, todos los elementos importados se accederán por su nombre original.

```
from paquete.modulo1 import *
```

```
print CONSTANTE_1
```

```
print CONSTANTE_2
```

Manejo de Objetos

Introducción a la orientación a objetos

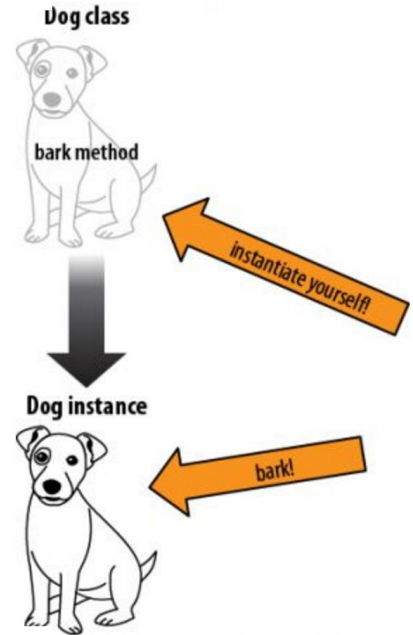
En Python todo es un objeto.

Un objeto es una instancia de una clase.

Las clases definen:

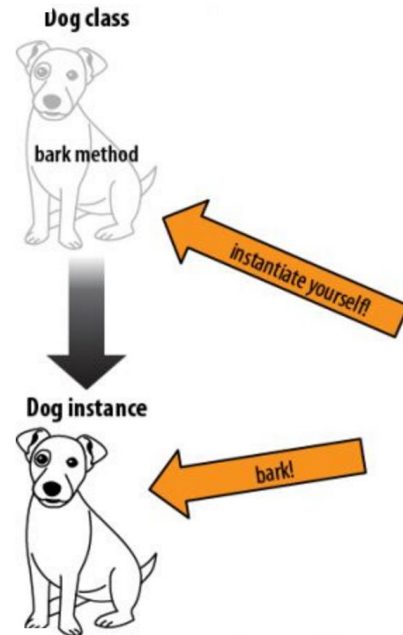
- Propiedades del objeto.

- Métodos (comportamiento).



Ejemplo:

```
1  # -*- coding: utf-8 -*-#
2  class Dog(): # Definición de clase
3      name = '' # <--- propiedad nombre
4      def bark(self): # <-- metodo para ladrar.
5          print "{}, guau... guau... ".format(self.name)
6          print
7
8  ''' Instancia de clase perro Doqui '''
9  perro1 = Dog()
10 perro1.name = 'Doqui'
11
12 ''' Instancia de clase perro Firulaes '''
13 perro2 = Dog()
14 perro2.name = 'Firulaes'
15
16 ''' Perros ladran '''
17 perro2.bark();
18 perro1.bark();
```



Acceso a propiedades públicas

```
1  # -*- coding: utf-8 -*-#
2  class Dog(): # Definición de clase
3      name = '' # <--- propiedad nombre
4      def bark(self): # <--- metodo para ladrar.
5          print "{}, guau... guau... ".format(self.name)
6          print
7
8  ''' Instancia de clase perro Doqui '''
9  perro1 = Dog()
10  perro1.name = 'Doqui'
11
12  ''' Instancia de clase perro Firulaes '''
13  perro2 = Dog()
14  perro2.name = 'Firulaes'
15
16  ''' Perros ladran '''
17  perro2.bark();
18  perro1.bark();
```

¿Y la
encapsulación?

Propiedades privadas

Python no cuenta con constructores (public, private o protected) para definir la visibilidad de las propiedades.

Para indicar que una propiedad o método es privado el nombre debe empezar con dos guiones bajos(__).

Las propiedades de la clase que inicien con dos guiones son privadas, lo que significa que no pueden accederse desde fuera de la misma.

```

1  # -*- coding: utf-8 -*-#
2  class Dog(): # Definición de clase
3      __name = '' # <--- propiedad nombre
4      def bark(self): # <--- metodo para ladrar.
5          print "{}, guau... guau... ".format(self.name)
6          print
7
8  ''' Instancia de clase perro Doqui '''
9  perro1 = Dog()
10 perro1.__name = 'Doqui'
11
12 ''' Instancia de clase perro Firulaes '''
13 perro2 = Dog()
14 perro2.__name = 'Firulaes'
15
16 ''' Perros ladran '''
17 perro2.bark();
18 perro1.bark();

```

¿Cual es el resultado de ejecutar este programa?

```

→ src git:(master) X python dogClassPrivateError.py
Traceback (most recent call last):
  File "dogClassPrivateError.py", line 17, in <module>
    perro2.bark();
  File "dogClassPrivateError.py", line 5, in bark
    print "{}, guau... guau... ".format(self.name)
AttributeError: Dog instance has no attribute 'name'

```

Código Fuente: [dogClassPrivateError.py](#)

Clase Dog con métodos getter & setters

```
1  # -*- coding: utf-8 -*-#
2  class Dog(): # Definición de clase
3      __name = '' # <--- propiedad nombre
4
5      def bark(self): # <--- metodo para ladrar.
6          print "{}, guau... guau... ".format(self.getName())
7          print
8
9      def setName(self, name):
10         self.name = name
11
12         def getName(self):
13             return self.name
14
15     ''' Instancia de clase perro Doqui '''
16     perro1 = Dog()
17     perro1.setName('Doqui')
18
19     ''' Instancia de clase perro Firulaes '''
20     perro2 = Dog()
21     perro2.setName('Firulaes')
22
23     ''' Perros ladran '''
24     perro2.bark();
25     perro1.bark();
```

Código Fuente: [dogClassSetterGetter.py](#)

Clase Dog con propiedades

```
1  # -*- coding: utf-8 -*-#
2  class Dog(): # Definición de clase
3      __name = '' # <--- propiedad nombre
4
5      def bark(self): # <-- metodo para ladrar.
6          print "{}, guau... guau... ".format(self.name)
7          print
8
9      @property
10     def name(self):
11         return self.name
12
13     @name.setter
14     def name(self, name):
15         self.name = name
16
17     ''' Instancia de clase perro Doqui '''
18     perro1 = Dog()
19     perro1.name = 'Doqui'
20
21     ''' Instancia de clase perro Firulaes '''
22     perro2 = Dog()
23     perro2.name = 'Firulaes'
24
25     ''' Perros ladran '''
26     perro2.bark();
27     perro1.bark();
```

Código Fuente: [dogClassProperty.py](#)

Comparando propiedad vs setter & getters

```
@@ -3,22 +3,24 @@ class Dog(): # Definición de clase
    __name = '' # <--- propiedad nombre

    def bark(self): # <--- metodo para ladrar.
-         print "{}, guau... guau... ".format(self.getName())
+         print "{}, guau... guau... ".format(self.name)
        print

-         def setName(self, name):
-             self.name = name
-
-         def getName(self):
+         @property
+         def name(self):
            return self.name

+         @name.setter
+         def name(self, name):
+             self.name = name
+
''' Instancia de clase perro Doqui '''
perro1 = Dog()
-perro1.setName('Doqui')
+perro1.name = 'Doqui'

''' Instancia de clase perro Firulaes '''
perro2 = Dog()
-perro2.setName('Firulaes')
+perro2.name = 'Firulaes'

''' Perros ladran '''
perro2.bark();
```

Guidelines

 El nombre de las clases debe ir en formato camel case, ejemplo:

Una clase figura se debe nombrar como: Figura.



Una clase Cuentacontable se nombra: CuentaContable.

Las propiedades se nombran con las mismas reglas de las variables.



Constructores

Al crear una instancia de un objeto es posible definir el estado inicial del objeto a través del constructor de la clase.

En Python un constructor es un método especial llamado `__init__`, el cual se invoca durante la instanciación del objeto, ejemplo:

```
def __init__(self, name):  
    self.name = name
```


Ejemplo constructor:

```
1  # -*- coding: utf-8 -*-#
2  class Dog(): # Definición de clase
3      __name = '' # <--- propiedad nombre
4
5      def __init__(self, name):
6          self.name = name
7
8      def bark(self): # <--- metodo para ladrar.
9          print "{}, guau... guau... ".format(self.name)
10         print
11
12     @property
13     def name(self):
14         return self.name
15
16     @name.setter
17     def name(self, name):
18         self.name = name
19
20     ''' Instancia de clase perro Doqui '''
21     perro1 = Dog('Doqui')
22
23     ''' Instancia de clase perro Firulaes '''
24     perro2 = Dog('Firulaes')
25
26
27     ''' Perros ladran '''
28     perro2.bark();
29     perro1.bark();
```

Código Fuente: [dogClassConstructor.py](#)

Implementando el constructor

```
class Dog(): # Definición de clase
    __name = '' # <--- propiedad nombre

+     def __init__(self, name):
+         self.name = name
+
    def bark(self): # <-- metodo para ladrar.
        print "{} guau... guau... ".format(self.name)
        print

@@ -15,12 +18,11 @@ class Dog(): # Definición de clase
    self.name = name

''' Instancia de clase perro Doqui '''
-perro1 = Dog()
-perro1.name = 'Doqui'
+perro1 = Dog('Doqui')

''' Instancia de clase perro Firulaes '''
-perro2 = Dog()
-perro2.name = 'Firulaes'
+perro2 = Dog('Firulaes')
+

''' Perros ladran '''
perro2.bark();
```

Ejercicio:

Elaborar un programa Python que lea "N" números, los almacene en una colección y despliegue la siguiente información ----->

Utilice un objeto para contener la lista y realizar las operaciones de mayor, sumatoria y promedio.

```
Ingrese el número (-1 salir): 1
Ingrese el número (-1 salir): 3
Ingrese el número (-1 salir): 5
Ingrese el número (-1 salir): 6
Ingrese el número (-1 salir): 7
Ingrese el número (-1 salir): 0
Ingrese el número (-1 salir): 8
Ingrese el número (-1 salir): -1
El mayor es 8
La sumatoria es 30
El promedio es 4.29
```

Bibliografía

Apuntes Instituto Tecnológico de Culiacán - MC. Emir Abel Manjarrez Montero

Python para principiantes [\(http://librosweb.es/libro/python/\)](http://librosweb.es/libro/python/) - Eugenia Bahit