

Python

Características Básicas

Intérprete

- `>>> 3 + 5`
- `8`
- `>>> "Esto es una" + " frase."`
- `Esto es una frase.`

 El **intérprete** de Python nos permite tener una sesión interactiva con él, muy útil para su aprendizaje.

Operaciones

- `>>> "Esto es una" + " frase."`
- `Esto es una frase.`




Con Python podemos realizar muchas operaciones, sean matemáticas o no.



Por ejemplo, 'sumar' dos textos produce su unión o concatenación. ¡Cuidado con los espacios!

Variables

```
• >>> hermanos = 3  
• >>> hermanos  
• 3
```

 Una *variable* no es más que un contenedor que almacena algún tipo de valor (numérico, texto...). Allí donde se use, su valor lo sustituirá.

 ¡Usa nombres descriptivos para tus variables!

Variables


- `>>> hermanos - 1`
- `2`
- `>>> saludo = "hola"`
- `>>> saludo * 3`
- `'holaholahola'`



Como en matemáticas, las variables pueden usarse en expresiones más complicadas.


Funciones

- `>>> len('Hola, Python')`
- `12`

 Otra similitud con las matemáticas son las **funciones**.
Ellas se encargan de realizar diferentes tareas.

 A menudo, devuelven como resultado un valor.


Módulos

 Un lenguaje de programación tiene unas funcionalidades básicas. Para extenderlas y hacerlo más potente, los programadores escriben librerías o *módulos*.

 Python viene con *pilas incluidas*.

import

- `import random`
- `sorteo = random.randint(1, 20)`


 Para cargar un módulo y poder usarlo se utiliza la instrucción **import**.

 Luego, puedes usar sus componentes utilizando la *notación dot*: **nombre_modulo.nombre_elemento**

Más Funciones


- `>>> import time`
- `>>> time.sleep(2)`


 La función *sleep()* del módulo *time* usa como **argumento o parámetro** un número.

 *sleep()* hace una espera de tantos segundos como indica el número que se le pase como argumento.

Definiciones


```
● >>> def saludar():  
●     ...     print "Saludos"  
●     ...  
● >>> saludar()  
● Saludos
```


 Puedes definir tus propias funciones con la *instrucción* **def** que inicia un bloque de definición.

 Cuando quieras que se ejecute la función, la invocas con su nombre (¡y los paréntesis!)

Argumentos

```
● >>> def hola(nombre):  
●     ...     print "Saludos, " + nombre  
●     ...  
● >>> hola('Susana')  
● Saludos, Susana
```


 Usando argumentos, podemos pasar valores a las funciones y operar con ellos.

 En la definición, los argumentos se indican con nombres y se usan como si fueran variables.

return

```
• >>> def suma(a,b):  
•     ...     return a + b  
•     ...  
• >>> 6 * suma(3,4)  
• 42
```

 Podemos usar la instrucción **return** en la definición para que la función devuelva un valor.

 De esta manera, cuando usemos la función, ésta queda sustituida por el valor devuelto por return.

Tipos de datos

 Números Enteros

 Números Decimales

 Cadenas de Texto

 Booleanos

 Tuplas

 Listas

 Diccionarios

 Números Enteros

● 27

● 1234567890L

 Números Decimales

● -27.36

 Cadenas de Texto

● 'Vaya toalla'

 Booleanos

● True

● False

 Listas

● [3 , 'calamar' , True]

 Tuplas


● (2.5 , 0 , -1e10)

 Diccionarios


● { 'Juan' : 3 , 'Pedro' : 5 , 'Ana' : 9 }


conversiones

- `>>> str(3.2)`
- `'3.2'`
- `>>> list("hola")`
- `['h', 'o', 'l', 'a']`

 Unos tipos de datos pueden convertirse en otros usando funciones de Python (cuyos nombres son precisamente el del tipo de dato al que se quiere convertir).

Listas

 En Python, las listas son objetos que almacenan una serie ordenada de otros objetos.

 Se construyen escribiendo, entre corchetes, los elementos deseados separados por comas.

Listas

- `lista = [4, 'hola', 3.5, True]`



Se puede acceder a los elementos de una lista indicando su índice. El primer elemento posee índice 0.


Listas

- `lista = [4, 'hola', 3.5, True]`

- `lista[0]`
 - `lista[3]`
 - `lista[-1]`
- 4
True
True

Slices

- `>>> perdidos = [4, 8, 15, 16, 23, 42]`
- `>>> perdidos[2:5]`
- `[15, 16, 23]`


 Una *slice* de una lista es un subconjunto de sus elementos. Se utiliza la notación : indicando dónde empezar y donde acabar (piensa en los huecos intermedios).


Slices

- `>>> perdidos = [4, 8, 15, 16, 23, 42]`
- `>>> perdidos[:5]`
- `[4, 8, 15, 16, 23]`
- `>>> perdidos[3:]`
- `[16, 23, 42]`
- `>>> perdidos[1:5:2]`
- `[8, 16]`
- `>>> perdidos[5:1:-2]`
- `[42, 16]`
- `>>> perdidos[::-1]`
- `[42, 23, 16, 15, 8, 4]`

Tuplas

```
>>> amigos = ( 'Juan' , 'Ana' , 'Olga' )  
>>> amigos[-2]  
'Ana'
```

 Una *tupla* es algo muy parecido a una lista y, de hecho, se trabaja de forma muy parecida.


 El matiz que las diferencia es que las tuplas son *inmutables*, es decir, **no pueden modificarse**.

Inmutabilidad

- `>>> lista = [4,8,15,16,23,42]`
- `>>> tupla = (4,8,15,16,23,42)`
- `>>> lista.append('perdidos')`
- `>>> lista`
- `[4,8,15,16,23,'perdidos']`
- `>>> lista.remove(8)`
- `>>> lista`
- `[4,15,16,23,'perdidos']`
- `>>> tupla.append('perdidos')`
- Traceback (most recent call last):
- File "<stdin>", line 1, in <module>
- AttributeError: 'tuple' object has no attribute 'append'

Diccionarios

- `>>> valores = {'a':10, 'b':11}`
- `>>> valores['a']`
- `10`
- `>>> valores['f'] = 15`
- `>>> valores`
- `{'a':10, 'b':11, 'f':15}`

 Los *diccionarios* permiten almacenar valores de forma no secuencial. Se puede acceder a ellos mediante sus correspondientes *claves hash* o *keys*

Diccionarios

- `>>> valores = {'a':10, 'b':11, 'f':15}`
- `>>> valores.keys()`
- `['a', 'b', 'f']`
- `>>> valores.has_key('b')`
- `True`
- `>>> valores['e']`
- `Traceback (most recent call last):`
- `File "<stdin>", line 1, in <module>`
- `KeyError: 'tuple'`
- `>>> valores.get('e')`
- `>>> valores.get('e', 'no existe')`
- `'no existe'`
- `>>> valores.get('a', 'no existe')`
- `10`


Polimorfismo


- `>>> len({'a':10, 'b':11})`
- `2`
- `>>> len('caracola')`
- `8`
-

 Tanto *listas* como *tuplas*, *diccionarios*... responden de forma similar ante diferentes métodos

Objetos

- `>>> "hola".upper()`
- `'HOLA'`

 En realidad, en Python todo son **objetos**, entes que tienen sus propiedades y sus comportamientos.

 Puede accederse a ellas de la misma forma que lo haceos con los módulos, con la *notación dot*.


Métodos

- `>>> "hola".upper()`
- `'HOLA'`
- `>>> "bye".islower()`
- `True`

 Los *métodos* gestionan su comportamiento; son funciones internas que permiten realizar diferentes tareas.

Atributos

- `>>> complejo = 2 + 3j`
- `>>> complejo.real`
- `2.0`

 Los *atributos*, por su parte, son el equivalente a valores o variables internas. Almacenan información útil.


Objetos y Clases

- 📌 Además, puedes definir tus propias **clases** de objetos. O importarlas, como veremos, desde otros módulos.

Pero este tema lo dejamos para más adelante...

Bloques


```
● i = 1
● while i < 3:
●     print "i vale", i
●     i = i + 1
● # El bucle ha terminado.
```


 Los *bloques* se indican con : y todos sus contenidos están sangrados (habitualmente, 4 espacios).

 Una vez que el bucle termina, el sangrado desaparece.

while

```
● i = 1
● while i < 3:
●     print "i vale", i
●     i = i + 1
```


 El bucle **while** ejecuta su contenido una y otra vez mientras se verifique la condición indicada.

 En el ejemplo, se mostrarán en pantalla los valores 1 y 2 de la variable i.

if ... elif ... else

```
• if 3 > 5:  
•     print "oro"  
• elif 3 = 5:  
•     print "plata"  
• else:  
•     print "bronce"
```


 El bucle **if** ejecuta su contenido si se cumple la condición indicada. Pueden usarse varias condiciones.

 En el ejemplo, se mostrará en pantalla el texto 'bronce'.

break


```
• while True:  
•     print "i vale", i  
•     i = i + 1  
•     if i == 20:  
•         break
```

 La instrucción **break** fuerza la salida de un bucle.


 En el ejemplo, observa la acumulación de sangrados y el uso de los símbolos `==` y `=`.


for ... in ...

- `for item in [1, 3, 5]:`
- `print "El item es", item`

 El bucle **for** recorre un iterable, por ejemplo una lista, y opera con cada uno de sus elementos.

Programas Independientes

 Un programa, guión o *script* en Python, es un **archivo de texto** con una serie de comandos que indican tareas que deben realizarse.

 Generalmente, los nombres de los archivos terminan en **.py** y son ejecutables

Ejemplo de Programa

- `#!/usr/bin/env python`
- `# -*- coding: utf-8 -*-`
- `print "¡Hola, Mundo!"`
- `raw_input()`


Autoejecución

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

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

- `print "¡Hola, Mundo!"`


- `raw_input()`

 Para que sea ejecutable, la primera línea ha de indicarle al sistema operativo con qué lenguaje está escrito.

 **¡Hay que darle permisos de ejecución al script!**

Codificación

- `#!/usr/bin/env python`
- **`# -*- coding: utf-8 -*-`**
- `print "¡Hola, Mundo!"`
- `raw_input()`

 Si utilizamos caracteres no anglosajones (acentos, eñes,...) debemos indicar que la codificación es internacional.

print

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

- **`print "¡Hola, Mundo!"`**

- `raw_input()`




La instrucción **print** muestra en pantalla el texto que le pasemos como argumento.

raw_input()

- `#!/usr/bin/env python`
- `# -*- coding: utf-8 -*-`
- `print "¡Hola, Mundo!"`

- **raw_input()**

 La función **raw_input()** espera a que el usuario pulse la tecla 'intro' y así nos permite ver el resultado en pantalla.

 ¡En caso contrario, se cierra la ventana al acabar!



etc



etc



etc



...

En cualquier caso...

¡Se aprende programando!