Python

Python es un lenguaje de programación de propósito general de alto nivel:

- Dado que el código se compila automáticamente y es ejecutado, Python es adecuado para su uso como lenguaje de implementación de aplicaciones web, o programación matemática, etc.
- Porque Python puede ampliarse en C y C++, Python puede proporcionar la velocidad necesaria, incluso para computar tareas intensivas.
- Debido a sus buenas construcciones de estructura (bloques de código anidados, funciones, clases, módulos y paquetes) y su uso consistente de los objetos y la programación orientada a objetos, Python nos permite escribir, aplicaciones lógicas claras para tareas pequeñas y grandes.

Importantes características de Python

- Tipos de datos incorporados: cadenas, listas, diccionarios, etc.
- Las estructuras de control habituales: if, if-else, if-elif-else, while, además del potente iterador (for).
- Varios niveles de estructura organizativa: funciones, clases, módulos y paquetes. Estos ayudan en la organización de código. Un ejemplo excelente es la librería estándar de Python.
- Compilar sobre la marcha el código de bytes. El código fuente es compilado a código de bytes sin un paso de compilación independiente.
- Python proporciona una forma consistente para utilizar objetos: todo es un objeto. Y en Python es fácil de poner en práctica nuevos tipos de objetos (llamados clases).

Importantes características de Python

- Extensiones en C y C++. Módulos de extensión usando herramientas como Swig.
- Jython es una versión de Python que funciona ’bien’ con Java. The Jython Project. Con Jython, podemos utilizar las bibliotecas y marcos de trabajo de Java.
- IronPython es una implementación para el entorno común de ejecución diseñado por Microsoft *(Common Language Runtime-CRL)*, más conocido como .NET. Con IronPython podemos utilizar todas las bibliotecas y estructuras CRL. IronPython.
- CPython, es la implementación más rápida, actualizada, sólida y más completa del lenguaje Python. CPython compila, interpreta y contiene módulos de extensión integrados y opcionales, todos codificados en C estándar.

Algunos enlaces importantes

- Todo el mundo Python esta aquí, Web de Python.
- El conjunto de documentación de Python estándar, Python documentation.
- Página de Ipython, un entorno interactivo de computación científica, Ipython interactive computing.
- Una serie de artículos, sobre la parte interna de Python, Python Internal.
- Artículos traducidos sobre el estilo de Python y mucho mas, Python idiomático.
- Una colección de lecturas de Python, sobre muchos temas, Lecturas de Python.

Corriendo Python

Después de haber instalado Python, puedes verificar tu instalacion, escribiendo en el terminal, la palabra 'python'. Si todo sale bien, verás algo parecido a esto

```
% python
Python 2.7.5+ (default, Sep 19 2013, 13:49:51)
[GCC 4.8.1] on linux2
Type "help", "copyright", "license" for more information.
>>> print "Python"
Python
```

Sobre Windows y Macintosh

- Python es lanzado como una aplicación.
- Una ventana con un interprete aparece y tu puedes ver el prompt.

Corriendo Python

Puedes, usar el entorno de Python, como un calculador, o puedes realizar alguna operación. Sugerimos fuertemente el uso de Ipython para hacer pruebas. El programa se termina al

- Tipear Control-D o Control-Z en el prompt interactivo.
- El programa corre hasta que EOF es alcanzado.
- o si escribimos

```
raise SystemExit
```

Convención-PEP8

Python utiliza indentación para mostrar la estructura de bloque. Indentación en el primer nivel muestra el inicio de un bloque. 'La proxima indentación fuera ' de la anterior muestra el final de un bloque. Por ejemplo:

```
if x:
    if y:
        f1()
    f2()
```

Y, la convención es usar cuatro espacios para cada nivel de indentación. En realidad, es más que una convención, es prácticamente un requisito.

Todos los cambios propuestos en Python, son detallados en documentos públicos llamados *Python Enhancement Proposals (PEP)*. Lo anterior de los espacios en blanco se encuentra en el PEP-8.

Sesiones interactivas

Cuando ejecutamos Python, sin un argumento de serie de comandos, Python inicia una sesión interactiva y nos indica que introduzcamos expresiones y sesiones de Python. Como por ejemplo, al introducir las siguientes declaraciones

```
>>> print "Hola␣a␣todos"
Hola a todos

>>> print  4 + 5
9

>>> print  True
True
```

Las sesiones interactivas son útiles para explorar, revisar y utilizar Python como una calculadora interactiva.

Entornos de desarrollo Python

- **IDLE**: Es el entorno de desarrollo Python, *(Integrated Development Environment) (IDLE)*. El IDLE, está basado en Tkinter. Incluye un editor de texto para escribir código fuente de Python, un depurador integrado integrado y otras propiedades interesantes....
- EL IDE modular, multiplataforma **ECLIPSE** tiene plugins que admiten Python, Jython, IronPython, Django, Unittest, etc. Ver pydev, para informarse más.
- Ipython es genial. IPython tiene un montón de cosas extras que ahorran una gran cantidad de tiempo y mejoran tu productividad. Tiene un editor de texto, que puede ser Vim, o algún otro, por ejemplo y usa 'funciones mágicas' para realizar las tareas.

Ejecutar programas en Python

Podemos ver nuestras aplicaciones como un conjunto de archivos fuente de Python, que son archivos de texto normales. Antes algunos conceptos:

Script Es un archivo que podemos ejecutar directamente.
Módulo Es un archivo que podemos importar para proporcionar funcionalidad a otros archivos o sesiones interactivas.

Los archivos fuente de Python, por lo general tienen extensión *.py*. Python guarda el archivo bytecode compilado para cada módulo que importamos. Python reconstruye cada archivo bytecode siempre que sea necesario.

(Por ahora) Podemos poner algo de código en un archivo hello.py, como esto

```
print "Hola a todos"
```

y ejecutarlo de la siguiente forma

```
% python hello.py
```

Lenguaje Python

Empezemos con algunas conceptos previos

- **Estructura léxica**: Conjunto de reglas que rigen la manera de escribir programas en Python.
- **Líneas y identación**: Un programa Python está compuesto por una secuencia de líneas lógicas, cada una compuesta por una o más línea físicas. Python utiliza la indentación para expresar la estructura de un programa.
- **Conjunto de caracteres** Los archivos Python están constituidos por carácteres ASCII. Pero le podemos decir a Python, que estamos utilizando un superconjunto de ASCII, solo en comentarios o cadenas literales, de la siguiente forma, por ejemplo

```
# -*- coding: iso-8859-1
```

Lenguaje Python

- **Símbolos** Cada línea lógica es una secuencia de componentes léxicos, llamados símbolos. Cada símbolo equivale a una subcadena de la línea léxica. Estos son
  - Identificadores: Nombre utilizado, para identificar una variable.
  - Palabras claves: Son identificadores que Python guarda para uso específico. Por ejemplo *(class, break, yield, pass . . .)*.
  - Operadores: Caracteres no-alfanuméricos y combinaciones de caracteres como operadores. Ejemplo, *(+, *, //, «, », !=, . . .)*
  - Separadores: Python usa símbolos y combinaciones como separadores de expresiones, listas, listas y diccionarios, cadenas. Por ejemplo (+=, *=, /=, //=, «=, »=, |=, :, ;, =, **=)
  - Literales: Es un número o una cadena de caracteres que aparece directamente en el programa. Utilizando separadores y literales, podemos crear valores de datos de clases fundamentales

```
[42, 3.8, 'python']      #Lista
(3,6, 9)                 #tupla
{'python':1, 'c++': 2}   #Diccionario
```

Lenguaje Python

- **Sentencias** Un archivo Python es como una secuencia simples y compuestas.
  - **Sentencia simple**: Es una sentencia que no contiene otra secuencia.

    ```
    a = 3
    ```

    Una asignación es una sentencia simple que asigna valores a las variables.
  - **Sentencia compuesta**: Contiene más secuencias y controla su ejecución. Una sentencia compuesta tiene una o más cláusulas alineadas en la misma identación. Cada cláusula tiene un encabezado que comienza con la palabra clave y finaliza con dos puntos (:), seguido por un cuerpo, que es una secuencia de dos o más sentencias. A esto se le llama *bloque*.

Tipos de datos en Python

El funcionamiento de un programa Python depende de los datos que gestiona. Todos los valores de los datos en Python son objetos y cada valor tiene un tipo. Un tipo de dato determina que operaciones admite el objeto, los atributos y los elementos del objeto y si el objeto puedo ser modificado o no.

Números

```
a = 3                   # Entero
b = 4.9                 # Punto flotante
c = 57484442227L        # Entero grande (Precision arbitraria)
d = 3 + 8j              # Numero Complejo
```

Secuencias

Una secuencia es un contenedor ordenado de elementos indexados por números enteros no negativos. Python proporciona tipos de secuencias integradas, denominadas cadenas (simples y Unicode), tuplas, y listas.

Tipos de datos en Python

Iterables

Todas las secuencias son iterables, es decir las listas son iterables, por ejemplo. Decimos que un podemos utilizar un iterable si a la larga dejará de producir elementos (iterable vinculado). Todas las secuencias se vinculan. Los iterables pueden ser 'desvinculados'.

Cadenas

```
a = 'Python'              # Una coma
b = "Optimizacion"    # Doble coma
c =  "Python tiene un modulo llamado 'Numpy'"
d ='''Una cadena de triple coma
puede escribirse  en multiples
lineas '''
e ="""Tambien trabaja para
comas dobles y todo funciona bien"""
```

Tipos de datos en Python

Cadenas

Una variante de la cadena literal es una cadena en bruto. La sintaxis es la misma que para cadenas entrecomilladas o con comillas triples, excepto que una r precede al entrecomillado principal. En cadenas en bruto las secuencias de salida como

```
\\              # Barra invertida
\<newline>      # Final de linea ignorado
\n              # Nueva linea
\r              # Retorno de carro
\f              # Avance
\b              # Retroceso
\t              # Tabulador
\DDD            # Valor octal
```

no son interpretadas de esta manera, sino que son copiadas literalmente en la cadena.

Tipos básicos en Python

Listas

```
a = [2,3,4]                    # Una lista de enteros
b = [2, 7, 3.5, "Python"]      # Una mix de objetos
c = []                         # Una lista vacia
d = [3, [a,b]]                 # Una lista de lista
```

Podemos usar el el tipo `list` para crear una lista. Por ejemplo

```
list('python ')              # ['p', 'y', 't', 'h', 'o', 'n']
```

Cuando x es iterable `list(x)` produce y devuelve una nueva lista cuyos elementos son los mismos que los elementos de x. Podemos crear comprensiones de lista con `list`.

Tipos básicos en Python

Tuplas

```
f = (3, 6, 7)          # Una tupla de enteros
g = (,)                # Una tupla vacia
h =(2, [3,4], (10,11,12))
```

Podemos usar el el tipo `tuple` para crear una tupla . Por ejemplo

```
tuple('jython')          # ('j', 'y', 't', 'h', 'o', 'n')
```

Cuando x es iterable `tuple(x)` devuelve una tupla cuyos elementos son los mismos que los elementos de x.

- Las tuplas son como las listas, pero el tamaño es fijado en el tiempo de creación.
- No podemos reemplazar los miembros de una lista (se dice que es 'inmutable').

Tipos básicos en Python

Una correspondencia es una colección de objetos indexados por valores 'casi' arbitrarios llamados claves. Las correspondencias son mutables y no están ordenadas. Python proporciona un único tipo de correspondencia, las de tipo diccionario. Un elemento en un diccionario es un par clave/valor.

Diccionarios

```
a = {}
b = {'Django': 1, 'Node.js' :2, 'R':3, }
c = {'uid' : 105,
 'login' : 'Lara',
 'name':  'Cesar Lara Avila'
 }
```

También podemos usar un tipo integrado llamado `dict` para crear un diccionario.

Tipos básicos en Python

Diccionarios

```
>> dict(x = 56, y = 56.7, z = 8 +3j)
{'x': 56, 'y': 56.7, 'z': (8+3j)}

>> dict ([[1,2], [3,4]])
{1: 2, 3: 4}

>> dict()
{}
```

Si el argumento x a `dict` es una correspondencia, `dict` devuelve un objeto del nuevo diccionario con las mismas claves y valores en x. Cuando x es iterable, los elementos en x deben ser pares y `dict(x)` un diccionario cuyos elementos son iguales a los elementos x.

Tipos básicos en Python

Diccionarios

También podemos llamar un diccionario llamando a `dict.fromkeys`. El primer argumento es un interable cuyos elementos se convierten en las claves del diccionario; el segundo argumento es el valor que corresponde a casa clave. Si se omite este último el valor que se le corresponde es `None`.

```
>>> dict.fromkeys('python',  4)
{'h': 4, 'o': 4, 'n': 4, 'p': 4, 't': 4, 'y': 4}

>>> dict.fromkeys([1,5,7])
{1: None, 5: None, 7: None}
```

Tipos básicos en Python

None

El tipo integrado None indica un objeto nulo.Las funciones por ejemplo devuelven None como su resultado sino se ha especificado setencias return codificadas para devolver otros valores o cuando señalamos que ’ningún objeto’ está aquí.

Invocables

Los tipos de datos que se pueden invocar (callable) son aquellos cuyas instancias aceptan la la función de "operación llamar". Las funciones son invocables. Los tipos de dato también son invocables (dict, list, tuple).Las clases y los métodos son invocables, que son funciones vinculadas a atributos de clase e instancias de clases que agregan un método __call__.

Tipos básicos en Python

Valores Booleanos

Cada valor de datos en Python, puede ser tomado como un valor de verdad; verdadero o falso.

El tipo integrado `bool` es una subclase de `int`. Los únicos dos valores de tipo `bool` son `True` y `False`, que tiene representaciones de cadena de "True", "False", pero también valores numéricos de 1 y 0 respectivamente.

El buen estilo Python recomienda escribir cualquier argumento x, como, por que es redundante

```
if x       # Nunca if bool(x); if x == True; y asi...
```

Variables y otras referencias

Un programa en Python accede a los valores de datos a través de las `referencias` que es un nombre que hace referencia a la localización en la memoria de un valor (objeto). Las referencias toman la forma de `variables`, `atributos` y `elementos`. Las variables en Python tienen las siguientes características

- Las variables son tipeadas dinámicamente (No tipificación explícita, los tipos pueden cambiar durante la execución).
- Las variables son sólo nombres para un objeto. No están atados a una localización de memoria como en C.
- El operador de asignación crea una asociación entre el nombre y el valor (vincular). Vincular una variable que ya esta vinculada, se llama "revincular" y no tiene consecuencia en el objeto que tiene vinculado, excepto cuando el objeto desaparece.
- Una variable puede ser global o local.

Variables y otras referencias

Elementos y atributos de objetos

La principal diferencia entre los atributos y los elementos en un objeto, es en la sintaxis que utilizamos para acceder a ellos.

```
objeto."nombre_atributo"        #Acceder a un atributo
x.y
```

Un elemento de un objeto se expresa con una referencia a un objeto, seguido de una expresión entre corchetes. La expresión entre corchetes se llama índice y el objeto se denomina contenedor del elemento.

```
objeto[]        #Acceder a un elemento
x[y]
```

Sentencias para las asignaciones

Las sentencias para las asignaciones pueden ser simples o aumentadas.

- **Asignación simple**

```
nombre = valor    # Asignacion para una variable
obj.attr = valor   # Asignacion al atributo de un objeto
obj[k] = valor    # Asignar a un elemento de un contenedor
obj[inicio: final: paso ]
```

Podemos introducir múltiples destinos y signos igual (=) en una asignación simple, puede también enumerar dos o más referencias separadas por comas (unpacking)

```
a = b = c = 0
a, b, c = x
a , b = b , a     # Intercambiar referencias
```

Sentencias para las asignaciones

- **Asignación aumentada** Se utiliza operadores aumentados como son: `+=, *=, «=, -=, ...`

  La diferencia entre objetos y referencia a un objeto es importante. Por ejemplo `x = x + y` no modifica el objeto al que el nombre x estaba originalmente vinculado. Más bien, revincula el nombre x para hacer referencia a un nuevo objeto. `x += y`, modifica el objeto al que el nombre x está vinculado cuando este objeto tiene el método especial `__iadd__`; de lo contrario, `x += y` revincula el nombre x a un nuevo objeto, algo así como `x = x + y`.

del

La sentencia `del` no borra objetos, desvincula referencias. La eliminación de objetos es una consecuencia automática.

```
del(ref)
```

Expresiones y operadores

Expresiones

Una expresión es un poco de código Python que evalúa para producir un valor. Las expresiones más simples son los literales e identificadores. Podemos crear otras expresiones uniendo subexpresiones y operadores y/o separadores.

```
3+4
3 ** 4
'Hello' +'Python'
3 * ( 4 + 5)
```

Operadores en expresiones

```
'expresion'          # Conversion de cadena
{ key: valor}        # Creacion de diccionario
[expre...]           # Creacion de una lista
(expre...)           # Creacion de una tupla
x[index: index1]     # Division
f(expr...)           # Llamada a funcion
x[index]             # Indexado
x.attr               # Referencia a un atributo
x ** y               # Elevar a la potencia
~x                   #Bit a bit NO
+x, -x               # Unario mas, menos
x * y, x / y, x // y, x % y
x + y, x - y
```

Operadores en expresiones

Continuación . . .

```
x << y, x >> y      # Mayuscula izquierda, derecha
x & y               # Bit a bit AND
x ^ y               # Bit a bit XOR
x | y               # Bit a bit OR
x < y, x <= y, x > y, x >=  y
x <> y, x != y, x == y
x is y, x is not y  # Prueba de identidad
x in y, x not in y  # Prueba de pertenencia
not x               # Booleano NOT
x and y             # Booleano AND
X or y              # Booleano OR
lambda arg,...expr  # Funcion simple anonima
```

Expresiones y operadores

Escribir en un archivo de texto con la extensión .py lo siguiente y ejecutarlo

```
principal = 1200     # Cantidad inicial
rate = 0.05          # Taza de interes
numyears = 5
year =1
while year <= numyears:
    principal = principal * (1 + rate)
    print year, principal
    year += 1
```

Y el resultado es.....

ejemplo2.py

Operaciones numéricas

Podemos realizar las siguientes operaciones

- **Conversiones numéricas** Se realiza una conversión explícita, transmitiendo el argumento numérico no complejo a cualquiera de los tipos integrados: `int, long, float, complex`.
- **Operaciones aritméticas**
  1. **División** El operador // realiza una división truncada. / realiza la división, devolviendo un resultado en punto flotante. Podemos usar `from __future__ import division` garantiza que el operador / funciones sin truncamiento.
  2. **Elevar a la potencia** `a**b` es la operación de elevar a la potencia. La función integrada `pow (a,b)` devuelve el resultado `a**b`.
  3. **Comparaciones** Tenemos los símbolos `<, <=, >, >=, ==, !=` que pueden ser usados para comparar dos enteros cualquiera.
- **Operaciones en números entero bit a bit** Los enteros y enteros largos, pueden ser considerados como una cadena de bits y utilizar todas las operaciones dadas anteriormente.

Operaciones en secuencia

- **Secuencias habituales** Funciones integradas `len, max, min, sum`.

```
z = [2,4, 6]
len(z) , max(z), min(z), sum(z)          # 3, 6, 2, 12
```

- **Conversiones en serie** Funciones integradas `tupla, list`. No existe una conversión implícita entre tipos de secuencia diferentes, excepto que las cadenas simples se transforman en cadenas Unicode.

```
x = (2,3, 5)
list(x)              # [2, 3, 5]
```

- **Concatenación y repetición** Usamos el operador + y el operador ∗.

```
'erika' +'cesar'      #'erikacesar'
3 * 'erika'           # 'erikaerikaerika'
```

Operaciones en secuencia

- **Evaluar la pertenencia** Operador `in` y `not in` comprueba si el objeto es cualquiera de la secuencia, devolviendo un valor booleano.

```
x = [4, 5, 6]
3 in x                    # False
```

- **Indexar una secuencia** El elemento `jth` de una sucesión S se indica como el indexado `S[j]`. Un índice negativo `j` indica el mismo elemento en S que hace `L + j`, donde S tiene L elementos.

```
x = ['python', 'C++', 'Android']
x[1] , x[-1]              # ('python',  'Android')
```

- **Dividir una secuencia** Podemos seleccionar una subdivisión con `s[i:j]`

```
x = [1,2,3,5]
x[1:3]                     #[2, 3]
```

Operaciones en secuencia

Cadenas

- Son inmutables.
- Los elementos de un objeto de cadena son cadenas, de longitud 1.
- Las divisiones de las cadenas son cadenas.

Tuplas

- Los objetos de una tupla no se pueden modificar.
- Los elementos de una tupla son objetos arbitrarios.
- Las divisiones de una tupla es una tupla.

Listas

- Los elementos de una lista se pueden modificar, por lo que podemos vincular o eliminar elementos o divisiones de una lista.
- Los elementos de una lista son objetos arbitrarios.

Modificar una lista

Manipulación de lista

```
x = a[1]                    # 2–elemento de la lista
y = b[1:3]                  # Retorna una sublista
z = d[1][0][2]              # Lista anidada
b[0] = 42                   # Cambia un elemento
```

Métodos de Lista

```
L.count(x)    # Numero de elementos de L que son iguales a x.
L.index(x)    # Indice del primer valor de L igual a x.
L.append(x)   # Agrega x al final de L
L.extend(s)   # Agrega elementos del iterable s al final de L.
L.pop(i)      # Elimina el valor de un elemento de L
              # en la posicion i.
```

Métodos de Lista

```
L.reverse()     # Revierte, en el lugar los elementos de L.
L.insert(i, x)  # Introduce el elemento x en L antes del
                # elemento en el indice i.
L.sort ()       # Ordena los elementos de L.
```

Ejemplo

```
>>> a = [3, 5, 7, 7, 'python']
>>> print a.count(7), a.count(5), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(11)
>>> a
[3, 5, -1, 7, 7, 11]
>>> a.sort()
>>> a
[-1, 3, 7, 7, 11]
```

Operaciones con diccionarios

Los diccionarios son contenedores y iterables y por ende se pueden realizar una variedad de operaciones. El hecho de que un diccionario sea iterable, permite transmitirlo a cualquier función o método que tome un argumento iterable.

- **Pertenencia al diccionario** Operador `in` comprueba si un operador es una clave de un diccionario. Devolviendo un valor booleano.

```
D = {'C++': 5, 'JavaScript': 3, 'Python': 4, 'kojo': 6}
'Python' in D
True
```

- **Indexar un diccionario** El valor en un diccionario `D` que está asociado con la clave `K` se indica mediante una indexación: `D[K]`. Indexar con una clave que no está presente en el diccionario, provoca una excepción.

Operaciones con diccionarios

Un ejemplo->

```
C = {'uid' : 105,
  'login' : 'Lara',
  'name':   'Cesar_Lara_Avila'
  }
u = C['uid']
C['shell'] = "/bin/zsh"  # Colocamos un nuevo elemento
if C.has_key("login"):   # Vemos si esta este elemento
    d = C['login']
else:
    d = None
```

Los objetos de diccionario proporcionan varios, métodos, algunos (no modificantes) que devuelven un resultado sin alterar el objeto al que se aplican, mientras que otros (modificantes) pueden modificar el objeto al que se aplican, como los hay en las lista.

Métodos de diccionarios

```
D.copy()     # Devuelve una copia superficial de D.
D.has_key(k) # Devuelve un valor booleano si la
              # clave del diccionario se encuentra o no.
D.items()    # Devuelve una lista de los elementos de D.
D.keys()     # Devuelve una lista con las claves de D.
D.values()   # Devuelve una lista con los valores de D.
```

Ejemplo

```
d = {'aa': 111, 'bb': 222}
d.keys()
['aa', 'bb']
d.values()
[111, 222]
d.items()
[('aa', 111), ('bb', 222)]
```

Métodos de diccionarios

```
D.iteritems()  # Devuelve un iterador con los elementos de D.
D.iterkeys()   # Devuelve un iterador con las claves de D.
D.itervalues() # Devuelve un iterador con los valores de D.
D.get(k[,x])   # Devuelve D[k] si k es una clave en D, de
               # de lo contrario devuelve x (o None ).
```

Ejemplo

```
d = {'aa': 111, 'bb': 222}
for key in d.iterkeys():
    print key

D = {'Python':4, 'JavaScript' :3, 'C++': 5, 'kojo': 6}
D.get('C++')
```

Métodos de diccionarios

```
D.clear()       # Elimina todos los elementos de D.
D.update(D1)    # Para cada k en D1, establece D[K]
                # igual a D1[k].
D.pop(k[,x])    # Elimina y devuelve D[k] si k es una clave.
D.popitem()     # Elimina y devuelve un elemento arbitrario.
D.setdefault(k[, x]) # Devuelve D[k] si k es una clave en D
                     # sino D[k] =x y devuelve x.
```

Ejemplo

```
w={"python":"web","c++":"optimizacion","javascript":"games"}
w1 = {"javascript.js":"servidor","R":"estadistica"}
w.update(w1)
print w
{'python': 'web', 'javascript.js': 'servidor', 'R':
 'estadistica', 'javascript': 'games', 'c++': 'optimizacion'}
```

Conjuntos en Python

Es un tipo de contenedor que ha sido parte de Python desde la versión 2.4. Un conjunto se compone de una colecciÃşn desordenada de objetos únicos e inmutables. Es una implementación de Python de los conjuntos, que se conocen en matemáticas. Un conjunto es iterable.

```
x = set("Python")
set(['h', 'o', 'n', 'P', 't', 'y'])

x = set(["Perl", "Python", "Java"])
set(['Python', 'Java', 'Perl'])

cities = set(("Paris", "Lima", "London","Berlin","Paris"))
set(['Paris', 'Lima', 'London', 'Berlin'])
```

Conjuntos en Python

Podemos definir conjuntos (desde python2.6) sin utilizar la función integrada `set`. Podemos utilizar llaves en su lugar:

```
lenguajes = {'python', 'c++', 'java', 'R'}
set(['python', 'R', 'java', 'c++'])
```

Los conjuntos se implementan de una manera, que no permite objetos mutables. El siguiente ejemplo demuestra, que no podemos incluir, por ejemplo, listas como elementos:

```
lenguajes = {(['python','ruby'], ['scala','c', 'visual'])}
.......
TypeError: unhashable type: 'list'
```

Los `Fronzensets` son como los conjuntos, excepto, que ellos son immutables.

Conjuntos en Python

```
cities = set(["Lima", "Paris","Roma"])
cities.add('Berlin')
set(['Roma', 'Paris', 'Berlin', 'Lima'])
```

```
cities = fronzeset(["Lima", "Paris","Roma"])
cities.add('Berlin')
....
AttributeError: 'frozenset' object has no attribute 'add'
```

Pertenencia al conjunto El operador `in` comprueba que si un objeto está en el conjunto. Devuelve un booleano. De igual modo `not in` .

Métodos de conjuntos

Los conjuntos proporcionan una serie de métodos. Los metodos no modificables devuelven un resultado sin alterar el objeto al que se aplican y se llaman `instancias de tipo frozenset`, mientras que los métodos modificables pueden alterar el objeto al que se aplican y se les llama `instancia de tipo set`.

```
S.copy()          # Devuelve una copia 'sombra'
S.difference()
S.intersection()
S.issubset(S1)
S.issuperset(S1)
S.symmetric_difference(S1)
S.union(S1)
```

Métodos de conjuntos

Ejemplo

```
x = {"a","b","c","d","e"}
y = {"b","c"}
x.difference(y)
set(['a', 'e', 'd'])

x1 = {"a","b","c","d","e"}
y1 = {"c","d"}
x1.issubset(y1)
False

cities1 = {"Bruselas","Brasilia","Rio␣Janeiro"}
cities1_backup = cities1.copy()
cities1.clear()
cities1.backup
set(['Rio␣Janeiro', 'Bruselas', 'Brasilia])
```

Métodos de conjuntos

```
S.add(x)        # Agrega x como un elemento de S.
S.clear(x)      # Elimina todos los elementos de x.
S.discard(x)    # Elimina x como un elemento de S.
S.pop (x)       # Elimina y devuelva un elemento de S.
S.remove(x)     # Elimina x como  un elemento de S.
                # Provoca una excepcion KeyErrorexception
                # si x no es un elemento de S
```

Ejemplo

```
x = {"a","b","c","d","e"}
x.pop()
'a'

x1 = {"a","b","c","d","e"}
x1.remove("z")
...
KeyError: 'z'
```

Sentencias de control de flujo

El flujo de control de un programa es el orden en que el código del programa se ejecuta. En Python esto se regula por `sentencias condicionales, bucles y llamadas de función`.

La sentencia if

La sentencia compuesta `if` (que consta de `if, elif` y cláusulas `else`) nos permite ejecutar condicionalmente bloques de sentencias.

```
if a == 'Python':
    op = 1
elif a == 'C++':
    op = 2
elif a = 'Node':
    op = 3
else:
    op = 4
```

Sentencias de control de flujo

La sentencia While

En Python `while` admite ejecuciones de una sentencia o bloque de sentencias que estén controladas por una expresión condicional.

```
import sys
text = ""
while 1:
    c = sys.stdin.read(1)
    text = text + c
    if c == '\n':
        break

print "Entrar: %s" % text
```

Sentencias de control de flujo

La sentencia for

La sentencia `for` en Python admite ejecuciones repetidas de una sentencia o bloques de sentencia, controladas por una expresión iterable.

```
for n in range (1,10):
    print "2␣a␣la␣potencia␣%d␣is␣%d" %(n, 2**n)
```

La sintaxis de la sentencia `for`:

```
for expr in iterable:
    declaraciones(s)
```

Sentencias de control de flujo

La sentencia for

`expr` es un identificador que nombra a la variable de control de bucle; la sentencia `for` sucesivamente 'revincula' esta variable a cada elemento del iterador, en orden. La sentencias que componen el cuerpo del bucle se ejecutan una vez para cada elemento en `iterable`. Podemos tener un destino con múltiples identificadores:

```
for key, value in d.items():
    if not key or not value:    # Mantener claves y valores
        del[key]
```

Realizar un bucle en una secuencia de números enteros es una tarea habitual, Python proporciona las funciones integradas `range` y `xrange`.

```
range(x)          # Lista de elementos desde 0 a x(excluido).
range(x,y)        # Lista de elementos desde x a y(excluido).
range(x,y, paso) # Lista de elementos desde x a y (excluido)
                  # con la diferencia de elementos subyacentes
                  # igual a paso.
```

Mientras que range devuelve un objeto de lista normal, xrange devuelve un objeto especial, dirigido al uso de interaciones con la sentencia `for`. Podemos usar ese iterador si queremos usando `iter(xrange())`.

```
>>print range(1,5)
[1,2,3,4]
>>print xrange(1,5)
xrange(1,5)
```

Compresión de lista

Muy a menudo, tendremos que construir una nueva lista de los elementos de una lista existente. La `Compresión de lista` puede ser útil. La sintaxis para la compresión de lista esta basada en set builder notation. Dada la forma `Y for X in LIST`, Y es referida a la función de salida, X es la variable, LIST es el conjunto de entrada. Esto dice hacer Y para cada X en LIST.

```
>>my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>[str(x) for x in my_list]
['1', '2', '3', '4', '5', '6', '7', '8', '9', '10']
```

Veamos otro ejemplo, queremos crear una lista de cadenas para cada elemento en `my_list`, excepto que esta vez sólo queremos los elementos que son números pares.

```
>>> n_list = []
>>> for x in my_list:
...     if (x % 2) == 0:
...         n_list.append(str(x))
...
>>> n_list
['2', '4', '6', '8', '10']

# Por comprension de lista

>>> n_list = [str(x) for x in my_list if (x % 2) == 0]
>>> n_list
['2', '4', '6', '8', '10']
```

Esta versión introduce una expresión después de la lista que actúa como un filtro en el que los elementos pasan a la función de salida.

Podemos iterar sobre más listas

```
>>> list_a = ['A', 'B']
>>> list_b = [4, 5]
>>> [(x, y) for x in list_a for y in list_b]
[('A', 4), ('A', 5), ('B', 4), ('B', 5)]
```

Lo que devuelve una tupla. Si desea listas anidadas, también puedes anidar una comprensión de lista dentro de otra.

```
>>> list_a = ['A', 'B']
>>> list_b = ['C', 'D']
>>> [[x+y for x in list_a] for y in list_b]
[['AC', 'BC'], ['AD', 'BD']]
```

Otras sentencias de Python

La sentencia break

Esta sentencia, sólo se permite dentro del cuerpo de un bucle. Cuando se ejecuta el break el bucle se finaliza. Veamos esto en el siguiente ejemplo que busca números primos:

```
>>> for n in range(2, 6):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'igual', x, '*', n/x
...             break
...     else:
...         print n, 'es un numero primo'
...
2 es un numero primo
3 es un numero primo
4 igual 2 * 2
5 es un numero primo
```

Otras sentencias de Python

La sentencia continue

Esta sentencia sólo se admite dentro del cuerpo del bucle. Cuando se ejecuta `continue` se ejecuta, la iteración actual del cuerpo del bucle finaliza y la ejecución continúa con la próxima interación del bucle.

```
>>> for num in range(2, 6):
...     if num % 2 == 0:
...         print "Numero par", num
...         continue
...     print "Un numero", num

Numero par 2
Un numero  3
Numero par 4
Un numero  5
```

Otras sentencias de Python

La sentencia pass

El cuerpo de una sentencia compuesta en Python no puede estar vacío; debe contener siempre una sentencia. Podemos utilizar una sentencia `pass` que no realiza ninguna acción, como marcador de posición cuando una sentencia es requerida.

```
if condicion1(x):
    proceso1(x)
elif condicion2(x) or x < 5:
    pass
else:
    proceso(x)
```

Las sentencias try y raise

Python admite la gestión de excepciones mediante la sentencia `try`, que incluye cláusulas `try`, `except`, `finally` y `else`.

Un programa puede provocar una excepción mediante la sentencia `raise`. cuando se provoca una excepción, el flujo de control del programa se detiene y Python busca un control de excepción apropiado.

Notas:

- Python no tiene una especial declaración *switch* o *case* para testear valores. Para manejar múltiples casos de test, usamos la declaración *elif*.
- La sentencia `while` y `for` pueden de forma opcional la cláusula `else`.

```
for x in container1:
    f is_ok(x): break    # el elemento x es correcto.
                         #  Finaliza el bucle.
else:
    print "No fue encontrado en el contenedor"
    x = None
```

Funciones

Las funciones son definidas con la sentencia `def`

```
def remainder (a,b):
    q = a//b
    r = a - q*b
    return r
```

El cuerpo de una función es una sucesión de sentencias que se ejecuta cuando la función es llamada. Tu puedes invocar a la función de la siguiente manera: `a = remainder (42,5)`. Puedes adjuntar los argumentos por defecto para los parámetros de la función mediante la asignación de valores en el definición de la función.

```
def testDefaultArgs(arg1, arg2='default2'):
    sentencias
```

Cuando una función define un parámetro con un valor por defecto, ese parámetro y todo los parámetros que siguen son opcionales. Si los valores no son asignados a todo parámetro opcional en la definición de la función, se produce una excepción SyntaxError.
funcion0.py

```
a = 2
def func1(x = a):
    return x

a = 3          # Reasignamos a
func1()        # Devuelve 2
```

Valores de los parámetros por defecto siempre se establecen en los objetos que se suministran como valores cuando se definió la función, como muestra el ejemplo anterior.

Lo anterior resulta dificil cuando el valor por defecto es un objeto modificable y el cuerpo de la función altera el parámetro

```
def f(x, y=[])
    y.append(x)
    return y
print f(34)     # [34]
print f(45)     # [34, 45]
```

Si queremos que y se vincule a un nuevo objeto de lista vacío, cada vez que se invoque f con argumento único, hacemos lo siguiente

```
def f(x, y=[])
    if y is None: y = []
    y.append(x)
    return y
print f(34)     # [34]
print f(45)     # [45]
```

Al final de los parámetros podemos utilizar cualquiera o ambas de las dos formas especiales `*args` y `**kwargs`. Cada llamada a la función puede vincular `args` a la tupla cuyos elementos son los argumentos posicionales extra. Del mismo modo `kwargs` se vincula a un diccionario cuyos elementos son los nombres y los valores de los argumentos nominales extra (keyword arguments). funcion1.py

```
def callfunc(*args, **kwargs):
    func(*args,**kwargs)
```

Este uso de `*args` y `**kwargs` es utilizado para escribir las envolturas y los proxies a otras funciones. Por ejemplo, `callfunc()` acepta cualquier combinación de argumentos y simplemente los pasa a través de `func()`.

Paso de parámetros y valores de retorno

[D.Beazley] Cuando se invoca una función, los parámetros de la función son simplemente nombres que hacen referencia a los objetos de entrada pasados. El significado de paso de parámetros no se ajusta correctamente a cualquier estilo único, como "paso por valor" o "pasar por referencia", como hay en otros lenguajes de programación. Por ejemplo, si se pasa un valor inmutable, el argumento parece, efectivamente, que se pasa por valor. Sin embargo, si es un objeto mutable (como una lista o diccionario) se pasa a una función donde luego es modificado, esos cambios se reflejan en el objeto original.

```
a = [1,2,3,4, 5]
def cuadrado(items):
    for i,x in enumerate(items):
        items[i] = x *x

cuadrado(a)     # Se cambia a [1,4,9,16, 25]
a
```

La sentencia Return devuelve un valor desde una función. Si el valor no es especificado o tu omites esta sentencia, el objeto None es retornado.

```
def divide(a,b):
    q = a//b
    r = a - q*b
    return r
x ,y =  divide(42, 5)  # x = 8, y = 2
```

Otro ejemplo, usando return

```
def f_multiple_lista():
  l1=[]
  l2=[]
  return (l1,l2)

def f0():
  list1 ,list2=f_multiple_lista()
  print list2
```