PmWiki Slideshow - running from PmWiki

Powered by S5

Lenguaje de Programación Python

Presentación para proyecto Butiá

Características Generales

- Lenguaje de Programación de *alto nivel* de *propósito general*.
- Énfasis en la *legibilidad* del código.
- Combina *gran poder* con *sintaxis muy clara y limpia*.
- Biblioteca estándar extensa y abarcativa.
- Multi paradigma: imperativo, orientado a objetos, funcional, reflectivo.
- Tipos dinámicos
- Manejo automático de memoria
- Lenguaje de *scripting* (no exclusivamente)

Historia

- Creado en 1989 por Guido van Rossum, en CWI (Holanda)
- Python 2.0: octubre 2000
- Puthon 2.6: octubre 2008.
- Python 3.0: diciembre 2008 (no compatible hacia atrás)

The Zen of Python

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
```

```
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
```

The Zen of Python(2)

```
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

http://www.python.org/dev/peps/pep-0020/

Spam and Eggs

- El nombre *Python* está basado en la serie *Monty Python's Flying Circus*.
- Sin embargo el logo hace referencia a la serpiente pitón

Algunas palabras clave:

- *pythonic* : Usa adecuadamente el lenguaje o está acorde con la filosofía python.
- *pythonistas*: admiradores y usuarios de python
- *pythoneers*: programadores python experimentados

Utilización de python

- Aplicaciones web. `mod_wsgi`, `mod_python`
- Lenguaje de scripting *embebido* en diferentes productos.
- Componente estándar de las útlimas distribuciones linux. Varios utilitarios están escritos en python.
- Sugar Software (OLPC) está escrito en python.
- Usuarios: Yahoo, YouTube, CERN, NASA.

Instrucciones

- asignación
- decisión: if-then-else

- iteración: for, while

Tipos de datos

- Numéricos:
  - enteros: `int`, `long`, `boolean`
  - reales: `float`
  - complejos: `complex`
- Cadenas: `string`
- Secuencias: listas, tuplas, cadenas,
- Otros: diccionarios, conjuntos

Sistemas de tipos

- Python es un lenguaje interpretado con tipado dinámico.
- Fuertemente tipado
- Las variables no tienen tipo, los objetos sí.
- En tiempo de ejecución se verifica que el tipo de los operadores se aplique a los operandos.
- El programador puede definir sus propios tipos (clases)

Duck typing

> "*when I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.*"

Ejemplo:

```
function calculate(a, b, c): return (a+b)*c

example1 = calculate (1, 2, 3)
example2 = calculate ([1, 2, 3], [4, 5, 6], 2)
example3 = calculate ('apples ', 'and oranges, ', 3)
```

Instrucción de Asignación

La instrucción de asignación tiene esta forma:

> *identificador* = *expresión*

- Asigna el valor a la variable correspondiente
- Si ya tenía valor, se pierde (asignación *destructiva*)
- Si no tenía valor, la variable se *crea* con la asignación (creación dinámica)
- Las variables no se declaran

Ejemplos de asignaciones

Algunos ejemplos de asignaciones en python:

```
altura = 0

base = 4.3 / 2.5

y = x * 1

cadena = "hola que tal"


contador = contador + 1
```

Sintaxis de la Asignación

La BNF correspondiente a la instrucción de asignación:

```
assignment_stmt ::=  (target_list "=")+ (expression_list | yield_expression)
target_list     ::=  target ("," target)* [","]
target          ::=  identifier
                     | "(" target_list ")"
                     | "[" target_list "]"
                     | attributeref
                     | subscription
                     | slicing
```

Asignación en cadena

Se puede asignar el mismo valor a muchas variables:

```
a = b = c = d = 1
```

es equivalente a

```
a = 1
b = 1
c = 1
d = 1
```

Asignación en Paralelo

Se pueden realizar varias asignaciones en paralelo:

```
a,b = 0,1
```

Asigna el valor 0 a `a` y el valor 1 a `b`

Es útil para intercambiar el valor de dos variables:

```
a,b = b,a
```

Notar que no es lo mismo que hacer:

```
a = b
b = a
```

La instrucción `if`

La instrucción `if` permite elegir entre dos bloques, de acuerdo con el resultado de una expresión:

```
if expresión:
   bloque-1
else:
   bloque-2
```

- `bloque-1` y `bloque-2` deben estar indentados con respecto a la línea anterior
- la sección `else` es opcional

Ejemplo de if

Un ejemplo de programa que utiliza un if para determinar si un número es par o impar:

```
# ingreso dato
numero = input('numero: ')

# determino si es par o impar

if numero % 2 == 0:
   resultado = 'par'
else:
   resultado = 'impar'

# mostrar resultado
print 'El numero', numero, 'es', resultado
```

Semántica del if

La ejecución de

```
if expresión :
   bloque-1

else :
   bloque-2
```

1. evalúa expresión

2. si el resultado es `True` ejecuta bloque-1
3. si el resultado es `False` ejecuta bloque-2

Booleanos en Python

Para python todo valor se puede interpretar como boolean:

- El valor False se asigna a:
  - Número 0 (de cualquier tipo)
  - cadena vacía
  - otros tipos (más adelante)
- El valor True se asigna a todos los otros
  - números no nulos
  - cadenas no vacías

En las clases se puede especificar la coerción a booleanos (método `__nonzero__`)

Selección en cascada: `elif`

La forma más general de la instrucción if:

```
if e1:
   S1
elif e2:
   S3
elif ...
   ...
elif en:
   Sn
else:
   S
```

Ejemplo

Determinar si un año es bisiesto

```
if anio % 400 == 0:
   print 'es bisiesto'
elif anio % 100 == 0:
   print 'no es bisiesto'
elif anio % 4 == 0:
   print 'es bisiesto'
else:
   print 'no es bisiesto'
```

Instrucciones de repetición:

- Python tiene dos instrucciones de repetición:
  - `while` : repetición controlada por una condición
  - `for`: repetición asociada con una estructura de *secuencia*

Listas

Una lista es una secuencia de valores, separados por comas, entre corchetes

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Listas: Indices y Rebanadas

- Al igual que las cadenas se puede acceder a elementos utilizando índices.
- También pueden utilizarse operadores: `+` y `*` y la función `len()`

```
>>> a[0]
'spam'

>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
```

Listas: mutabilidad

A diferencia de las cadenas, las listas son *mutables*

```
>>> a
['spam', 'eggs', 100, 1234]

>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Asignación con slices

También se pueden *cambiar* porciones de una lista:

```
>>> # Replace some items:
... a[0:2] = [1, 12]

>>> a
[1, 12, 123, 1234]
>>> # Remove some:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Insert some:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
```

La instrucción `for`

- Sirve para *recorrer* todos los elementos de una secuencia y ejecutar un bloque asociado con cada elemento

```
>>> a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
```

Sintaxis de `for`

La instrucción for tiene esta forma:

```
for variable in secuencia :
          bloque
```

Semántica de `for`

La ejecución de:

```
for v in [x1,...,xn] : instrucción
```

es equivalente a ejecutar:

```
v = x1; instrucción
v = x2; instruccion
...
v = xn; instrucción
```

La función `range()`

Permite construir una secuencia de valores consecutivos:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Iterar sobre los índices de una secuencia

- Muy frecuente en lenguajes como C, Java: recorrer un arreglo a través de sus índices
- Poco frecuente en lenguajes de scripting (no muy *pythonico*)

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```

Un clásico: factorial

- Calcular el factorial de un número N:

```
N = input('Numero: ')
factorial = 1
for i in range(2,N+1):
    factorial *= i
print 'el factorial de', N, 'es', factorial
```

La instrucción `while`

La instrucción while permite repetir la ejecución de un bloque *mientras* una condición se mantiene verdadera:

```
# calcular factorial de un número N
N = input('numero: ')
factorial = 1
contador = 1
while contador <= N:
    factorial = factorial * contador
```

```
    contador = contador + 1
print 'El factorial de', N, 'es', factorial
```

Sintaxis y semántica de while

La instrucción while tiene esta forma:

while *expresión* **:**
bloque

1. se evalúa la expresión
2. si el resultado es `True` se ejecuta *bloque* y se retorna al paso 1
3. si el resultado es `False`, termina la ejecución

Observar que bloque puede no ejecutarse nunca

Ejemplo: número primo

Determinar si un número es primo:

```
from math import *
numero = input('Numero: ');
fin = floor(sqrt(numero))
divisor = 2
while (divisor <= fin) and (numero % divisor != 0) :
   divisor = divisor + 1
if divisor <= fin:
   print 'El numero', numero, 'no es primo'
else:
   print 'El numero', numero, 'es primo'
```

break

```
from math import *
numero = input('Numero: ');
fin = floor(sqrt(numero))
for divisor in xrange(2,fin+1):
   if numero % divisor == 0 :
      print 'El numero', numero, 'no es primo'
      break
else:
   print 'El numero', numero, 'es primo'
```

Secuencias

En python se consideran secuencias varios tipos distintos:

- listas
- cadenas
- tuplas

Todos ellos tienen operaciones en común

Operaciones de secuencia

Operaciones comunes a todos los tipos secuenciales

- `x in s`: True si algún ítem de s es igual a x
- `x not in s` False si algún ítem de s es igual a x
- `s + t` concatenación de s y t
- `s * n` n copias de s concatenadas
- `s[i]` i-ésimo ítem de s
- `s[i:j]` rebanada de s desde i hasta j
- `s[i:j:k]` rebanada con paso k
- `len(s)` largo de s
- `min(s)` mínimo de s
- `max(s)` máximo de s

Tuplas

Son muy similares a las listas, pero son inmutables

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)

>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Métodos para listas

Los métodos son operaciones especiales (más adelante: clases)

- `list.append(x)` Agrega x al final de la list
- `list.extend(L)` Agrega los elementos de L al final de list
- `list.insert(i, x)` Inserta x en la posición i
- `list.remove(x)` Borra la primera aparición de x en la lista. Produce un error si x no aparece.
- `list.pop(i)` Borra el ítem en la posición i y retorna su valor. Si no se especifica i, borra el último.

Métodos para listas (cont)

- `list.index(x)` Retorna el índice del primer ítem de la lista cuyo valor es x. Produce un error si x no aparece.
- `list.count(x)` Retorna el número de veces que x aparece en la lista.
- `list.sort()` Ordena los elementos de la lista.
- `list.reverse()` Invierte los elementos de la lista.

Métodos para listas. Ejemplos

Ejemplos de invocación de métodos para listas:

```
>>> a = [66.25, 333, 333, 1, 1234.5]

>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]

>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

Listas por comprensión

Es una manera de crear listas a partir de otras secuencias Es similar a la definición de conjuntos por comprensión

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [[x, x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

Listas por comprensión (2)

Es posible usar un método para construir los elementos de la nueva lista

```
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
```

Se puede filtrar algunos elementos con if

```
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
```

Listas por comprensión (3)

Es posible combinar dos o más listas:

```
>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

La instrucción `del`

Sirve para borrar un elemento de una lista según su índice. Es similar a `pop` pero no retorna el valor

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]

>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]    #  borrar una rebanada
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
>>> del a # borrar la lista completa
```

Conjuntos

- Python incluye un tipo de datos para representar conjuntos.
- Los conjuntos no contienen elementos repetidos y sus elementos no tienen un orden secuencial

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruit = set(basket)               # create a set without duplicates
```

```
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit                 # fast membership testing
True
>>> 'crabgrass' in fruit
False
```

Conjuntos. Operaciones

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                 # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                             # letters in a but not in b
set(['r', 'd', 'b'])
>>> a | b                             # letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                             # letters in both a and b
set(['a', 'c'])

>>> a ^ b                             # letters in a or b but not both
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

Diccionarios

Los diccionarios representan una correspondencia entre un conjunto de claves y un conjunto de valores. Las claves son únicas y pueden ser de cualquier tipo inmutable (generalmente son string)

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098

>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

Diccionarios desde listas

La función `dict()` construye un diccionario a partir de una lista de pares:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in (2, 4, 6)])     # use a list comprehension
{2: 4, 4: 16, 6: 36}
```

Notación simplificada

Resulta más sencillo invocar la función dict() de la siguiente manera:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

Operaciones con diccionarios

- `len(d)` Retorna la cantidad de ítems en el diccionario d
- `d[key]` Retorna el ítem de d con clave key. Retorna un error si no existe tal ítem.
- `d[key] = value` Asigna el valor value a la clave key
- `del d[key]` Borra d[key] de d
- `key in d` Retorna True si d tiene una clave key
- `key not in d` equivalente a not key in d
- `d.items()` Retorna una lista de pares (clave,valor)
- `d.keys()` Retorna una lista de las claves en el diccionario
- `d.values()` Retorna una lista de los valores en el diccionario

Recorrida de un diccionario

Se utilizan los iteradores: iteritems(), iterkeys(), itervalues()

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave
```

Subprogrmas. Introducción

- Un subprograma es un fragmento de código que se comporta de manera *independiente* dentro de un programa.
- Los subprogramas pueden ser invocados varias veces desde otras partes del programa.

Se comunican mediante el pasaje de parámetros.

Cada subprograma tiene su propio espacio de nombres (identificadores locales)

Algunos identificadores pueden ser compartidos entre subprogramas y el programa principal (identificadores globales).

Los subprogramas son una herramienta de modularización.

Ejemplo de Función

Una vez más: el factorial de un número:

```
def factorial(num):
        "calcula el factorial de num"
        facto = 1
        for i in xrange(2,num+1):
                facto *= i
        return facto
```

- `num` es un *parámetro* o *argumento*
- `facto`, `i` son variables *locales*

Cómo se utiliza una función

Un programa que utiliza la función factorial

```
def factorial(num):
        facto = 1
        for i in xrange(2,num+1):
                facto *= i
        return facto

for a in xrange(1,11):
    print a,'! = ', factorial(a)
```

Sintaxis de la definición de funciones

Una definición de función tiene esta forma general:

def *nombre* **(** *argumentos* **)**:
bloque de instrucciones

- *nombre* es un identificador
- *argumentos* es una lista de identificadores
- *bloque* debería contener una instrucción `return`

La instrucción `return`

Tiene esta forma:

return *expresión*

- sólo puede utilizarse en funciones
- termina la ejecución de una función
- retorna el valor de la expresión
- si una función no tiene `return` el valor que retorna es `None`

Ejemplo. Números primos

Nuevamente: Obtener los primos entre 0 y un número dado N

```
def divide(a,b):
  "determina si a es divisor de b"
  return (b % a == 0)

def AlgunoDivide(nums,Num):
   """determina si algún número de la lista nums
      es divisor de Num """
   for i in nums:
      if divide(i,Num):
         return True
   return False
```

Ejemplo (continuación)

La función principal utiliza `AlgunoDivide`

```
def primos_hasta(Numero):
        primos = []
        for n in xrange(2,Numero+1):
                if not AlgunoDivide(primos,n):
                        primos.append(n)
        return primos
```

Argumentos por omisión

- Ciertos argumentos efectivos pueden suprimirse y se asocia un valor por *omisión*

```
def primos_hasta(Numero=50):
   ...
```

- El parámetro `Numero` tiene valor `50` por omisión.
- Entonces la función puede invocarse sin parámetro efectivo

```
>>> primos_hasta()
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Argumentos por clave

- Es otra modalidad de pasar los argumentos a una función:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

- Esta función se puede invocar de todas estas maneras:

```
parrot(1000)
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

Programación Funcional

Programación Orientada a Objetos

Módulos y paquetes

Bibliografía

- Sitio web: http://www.python.org/
- Tutorial - Introduce informalmente los conceptos y propiedades básicas del lenguaje Python.
- Bibliotecas - Describe todas las funciones y métodos de la librería estándar de python
- Referencia del Lenguaje - Describe con precisión la sintaxis y semántica del lenguaje Python