Python: Funciones, Tipos de Variables y Recursividad

Herramientas Computacionales

fl.gomez10 at uniandes.edu.co

2015-2

Tipos de variables en Python

En python se tienen cinco tipos básicos de variables:

- Números
- Strings o cadenas de caracteres.
- Listas
- Tuplas
- Diccionarios

Números

Existen cuatro tipos de números en python:

- **int** Plain integers. Enteros con signo; 0, 1, 2, 3, …, -1, -2, -3, … Precisión: 32bits
- **long** Enteros con precisión ilimitada, se especifican con una “L” al final; 68476315848534846 3L, 99999988965465213213506451 0684L
- **float** Floating Point, números decimales, la precisón interna depende de la máquina; 3.14165, -0.99
- **complex** Números complejos, parte real e imaginaria, se especifican como floats; 3.0 + 4.0j

https://docs.python.org/2/library/stdtypes.html

Operaciones con Números

Operation	Result
`x + y`	sum of *x* and *y*
`x - y`	difference of *x* and *y*
`x * y`	product of *x* and *y*
`x / y`	quotient of *x* and *y*
`x // y`	(floored) quotient of *x* and *y*
`x % y`	remainder of `x / y`
`-x`	*x* negated
`+x`	*x* unchanged
`abs(x)`	absolute value or magnitude of *x*
`int(x)`	*x* converted to integer
`long(x)`	*x* converted to long integer
`float(x)`	*x* converted to floating point
`complex(re,im)`	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero.
`c.conjugate()`	conjugate of the complex number *c*. (Identity on real numbers)
`divmod(x, y)`	the pair `(x // y, x % y)`
`pow(x, y)`	*x* to the power *y*
`x ** y`	*x* to the power *y*

https://docs.python.org/2/library/stdtypes.html

str

Cadenas de caracteres. Son objetos tipo secuencia modificables. Se definen siempre entre comillas sencillas o dobles;

“Hola mundo!”

‘Hola mundo!’

Métodos más comunes: find, count, split, splitlines, translate.

https://docs.python.org/2/library/stdtypes.html

list

Listas. Son objetos tipo secuencia modificables, funcionan como contenedores que pueden almacenar distintos tipos de objetos. Se escriben siempre entre corchetes cuadrados.

desayuno = [“leche”, “huevos”, “pan”]

notas = [4.5, 6.7j, “no presentó”, desayuno, 6.022*10**-23]

https://docs.python.org/2/library/stdtypes.html

Operaciones sobre objetos tipo secuencia modificables

Operation	Result	Notes
`s[i] = x`	item *i* of *s* is replaced by *x*	
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*	
`del s[i:j]`	same as `s[i:j] = []`	
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*	(1)
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list	
`s.append(x)`	same as `s[len(s):len(s)] = [x]`	(2)
`s.extend(x)`	same as `s[len(s):len(s)] = x`	(3)
`s.count(x)`	return number of *i*'s for which `s[i] == x`	
`s.index(x[, i[, j]])`	return smallest *k* such that `s[k] == x` and `i <= k < j`	(4)
`s.insert(i, x)`	same as `s[i:i] = [x]`	(5)
`s.pop([i])`	same as `x = s[i]; del s[i]; return x`	(6)
`s.remove(x)`	same as `del s[s.index(x)]`	(4)
`s.reverse()`	reverses the items of *s* in place	(7)
`s.sort([cmp[, key[, reverse]]])`	sort the items of *s* in place	(7)(8)(9)(10)

tuple

Tuplas. Son objetos tipo secuencia no modificables, de solo lectura, funcionan como contenedores que pueden almacenar distintos tipos de objetos. Se escriben siempre entre paréntesis.

cena = (“leche”, “huevos”, “pan”)

datos = (4.5, 6.7j, “no presentó”, cena, 6.022*10**-23)

https://docs.python.org/2/library/stdtypes.html

dict

Diccionarios.

A diferencia de las listas, en los diccionarios los elementos no tienen un orden. Almacenan pares “clave-valor” (key-value). Se declaran entre corchetes “{ }”, pueden almacenar números, strings o tuplas. No pueden almacenar listas.

dict = { ‘name’:’Felipe’, ‘age’:28, ‘country’:’Colombia’}

superseleccion = { ‘Cordoba’:1, ‘Escobar’:2, ‘Valderrama’:10, ‘Rincon’:19

‘Asprilla’:21, ‘Alvarez’:14, ‘Valencia’:11}

https://docs.python.org/2/library/stdtypes.html

métodos en dict

has_key(k) busca si la clave k está en el diccionario.

del dict[n] permite borrar elementos individuales por su clave n

clear() borra todos los elelmentos de un diccionario.

keys() devuelve todas las claves en el diccionario

values() devuelve todos los valores en el diccionario

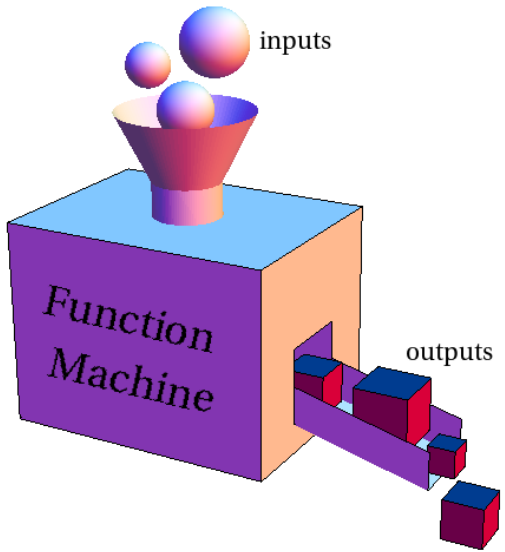
items() devuelve todas las parejas clave-valor

get() busca la clave en el diccionario y devuelve el valor

pop(k) borra la clave k del diccionario y revuelve su valor.

http://pycol.blogspot.com.co/2009/03/diccionarios-en-python.html

Funciones



Funciones

La palabra “def” está reservada para definir funciones. Debe ir seguida del nombre de la función y entre paréntesis los argumentos o parámetros de entrada con los que operará la función. Cerrando la línea van dos puntos.

```
def funcion( argumentos ):
    """documentacion opcional entre tripletas de comillas"""
    # comandos
    # comandos

    return objeto(s) a devolver
```

Funciones

La primera línea de una función puede ser un docstring, o línea de documentación. Esto es muy importante cuando se trabaja en proyectos grandes.

Siguen los comandos a ejecutar. Cada línea debe ir indentada, preferiblemente con cuatro (4) espacios en blanco, o con tabulado.

```
def funcion( argumentos ):
    """documentacion opcional entre tripletas de comillas"""
    # comandos
    # comandos

    return objeto(s) a devolver
```

Funciones

Para cerrar la función va la palabra “return” seguida de los objetos que devuelve la función. Generalmente son números, pero pueden ser listas, strings, etc.

```
def funcion( argumentos ):
    """documentacion opcional entre tripletas de comillas"""
    # comandos
    # comandos

    return objeto(s) a devolver
```

Funciones

Para cerrar la función va la palabra “return” seguida de los objetos que devuelve la función. Generalmente son números, pero pueden ser listas, strings, etc.

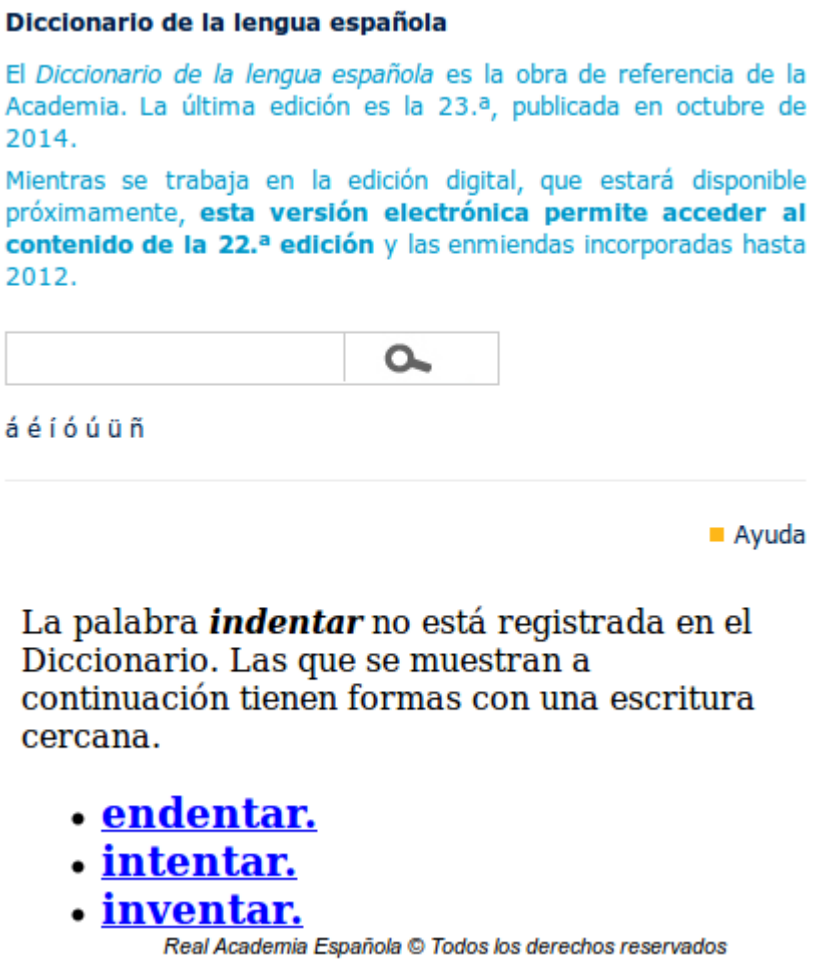
```
def funcion( argumentos ):
    """documentacion opcional entre tripletas de comillas"""
    # comandos
    # comandos

    return objeto(s) a devolver
```

(Paréntesis de la RAE)

“indentación” es un anglicismo no aceptado por la Real Academia Española.

Se recomienda utilizar “sangrado” o “sangría”, términos de buen uso en la industria de la impresión.

REAL ACADEMIA ESPAÑOLA



hola mundo!

Podemos tener funciones que no requieran argumentos de entrada y tampoco retornen salida.

```
def hola():
    print "Hola mundo!"
```

```
hola()
```

```
Hola mundo!
```

Funciones

funciones con números como parámetros de entrada y números como valor de retorno.

```
def suma_de_cuadrados(x, y):
    aux = x**2 + y**2
    return aux
```

```
suma_de_cuadrados(3, 4)
```

```
25
```

Funciones

… O funciones con

diversos argumentos

de entrada.

```python
def calificacion( p1, p2, p3, nombre):
    prom = (p1 + p2 + p3)/3.0
    if (prom>= 3.0):
        texto = "Aprobado"
    else:
        texto = "Reprobado"
    return nombre, prom, texto
```

```python
calificacion(5.0, 5.0, 5.0, "Heriberto de la Calle" )
```

```
('Heriberto de la Calle', 5.0, 'Aprobado')
```

```python
type(calificacion(5.0, 5.0, 5.0, "Heriberto de la Calle" ))
```

```
tuple
```

Funciones

… O funciones que

llamen otras

funciones.

```
def saludo():
    hola()
    print suma_de_cuadrados(1,1)
```

```
saludo()
```

```
Hola mundo!
2
```

Fibonacci, de nuevo.


```
>>> def fib(n):    # escribe la serie de Fibonacci hasta n
...     """Escribe la serie de Fibonacci hasta n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Ahora llamamos a la funcion que acabamos de definir:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Fibonacci, de nuevo.

Las funciones también pueden cambiar de nombre.

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Fibonacci, la venganza.

Ahora vamos a tener una función que retorne una lista.

```
>>> def fib2(n): # devuelve la serie de Fibonacci hasta n
...     """Devuelve una lista conteniendo la serie de Fibonacci hasta n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)     # ver abajo
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)     # llamarla
>>> f100                 # escribir el resultado
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Recursividad

El poder de la recursividad evidentemente se fundamenta en la posibilidad de definir un conjunto infinito de objetos con una declaración finita. Igualmente, un número infinito de operaciones computacionales puede describirse con un programa recursivo finito, incluso en el caso de que este programa no contenga repeticiones explícitas.

Wirth, Niklaus (1976)

Recursividad

Una función puede llamarse a sí misma para resolver un problema.

¿Cómo escribir un procedimiento recursivo?

- Entender qué es lo que tiene que hacer el proceso.
  - cuáles son los parámetros de entrada (inputs)
  - qué debe retornar el proceso (outputs)
- Pensar en algunos ejemplos de entradas y salidas
- Definir el procedimiento (siguiente diapositiva)
- ¡Probar!

¿Cómo definir una función recursiva?

1) Definir el caso base:

La respuesta más simple que puede devolver nuestra función. Para números enteros, generalmente el caso base es cero como parámetro de entrada.

2) Definir los pasos recursivos:

Pensar qué debe hacerse de manera repetitiva para reducir al caso base y finalizar el proceso.

Esqueleto de una función recurrente

```
def: mi_funcion( mi_input):

    if (es el caso base?):

        return respuesta_caso_base( mi_input)

    else:

        return combinar( mi_imput , mi_funcion( acercar_mi_input_al_caso_base ) )
```

Ejemplo: Función Factorial

Tenemos esta definición matemática de la función factorial para números enteros.

0! = 1

n! = 1 x 2 x … x (n-1) x n

```
def factorial(n):
    if (n==0):
        return 1
```

```
factorial(0)
```

```
1
```

Arrancamos por nuestro caso base:

y funciona!

Ejemplo: Función Factorial

0! = 1 --------> OK

Ahora vemos cómo escribir el problema en forma recursiva:

n! = 1 x 2 x … x (n-1) x n

n! = (n-1)! x n

definiendo un factorial en términos de otro factorial.

```
def factorial(n):
    if (n==0):
        return 1
    else:
        return n*factorial(n-1)
```

```
factorial(3)
```

```
6
```

Ejemplo: Función Factorial

Al definir un factorial en términos de otro factorial se generan pasos recursivos hasta llegar al caso base.

3! = 2! x 3

2! = 1! x 2

1! = 0! x 1

0! = 1

1! = 1 x 1 = 1

2! = 1 x 2 = 2

3! = 2 x 3 = 6

```
def factorial(n):
    if (n==0):
        return 1
    else:
        return n*factorial(n-1)
```

```
factorial(3)
```

```
6
```

(Paréntesis de la RAE)

Las palabras murciégalo, toballa, crocodilo, almóndiga y otros bichos raros han sido aceptados por la RAE.

U_U

REAL ACADEMIA ESPAÑOLA