Módulo de adaptación

Master en Business Intelligence y Big Data

PROFESOR/A
Antonio Sarasa Cabezuelo

Escuela de organización industrial

www.eoi.es

Introducción a Python(I)

Índice

Introducción

- Python es un lenguaje de programación de alto nivel que se caracteriza por:
  - Es simple
  - Es fácil de leer y escribir
  - Es fácil de depurar.
  - Portable

Introducción

- Python es un lenguaje **interpretado**:
  - Los **interpretes** leen programas de alto nivel y los ejecutan comando a comando.
  - Los **compiladores** leen el programa entero (código fuente) antes de que el programa empiece a ejecutarse y convierte el código fuente en un ejecutable que puede ser ejecutado tantas veces como se quiera sin necesidad de volverlo a traducir.
  - Los **Interpretes** proporcionan feedback instantaneo.
  - Algunas desventajas son el hecho de que son más lentos y puede que ocupen más espacio.

Variables

- Una variable es un nombre que referencia un valor.

```
In [2]: titulo = "¿Cómo encontrar el área de un círculo?"
        pi = 3.14159
        radio = 5
        area = pi * (radio**2)

In [3]: print area

        78.53975
```

Variables

- Una sentencia de asignación crea variables nuevas y las da valores:

```
In [1]: mensaje = 'Y ahora algo completamente diferente'

In [2]: n = 17

In [3]: pi = 3.1415926535897931
```

Variables

- Para mostrar el valor de una variable, se puede usar la sentencia print:

```
In [4]: print n
        17

In [5]: print pi
        3.14159265359
```

Variables

- Las **Variables** son de un tipo, que coincide con el tipo del valor que referencian.

```
In [6]: type(mensaje)
Out[6]: str

In [7]: type(n)
Out[7]: int

In [8]: type(pi)
Out[8]: float
```

Variables

- Algunos de los tipos más usados son:
  - int :enteros
  - float :números reales
  - bool :valores booleanos: cierto y falso
  - str :cadenas
  - None :corresponde al valor nulo

```
In [1]: type(4)
Out[1]: int

In [2]: type(3.14159)
Out[2]: float

In [3]: type('3.14159')
Out[3]: str
```

Variables

- Reglas de construcción de los nombres de las variables:
  - Pueden ser arbitrariamente largos
  - Pueden contener tanto letras como números
  - Deben empezar con letras.
  - Pueden aparecer subrayados para unir múltiples palabras.
  - No pueden ser palabras reservadas de Python

Variables

- Algunos ejemplos:

```
In [4]: type(area)
Out[4]: float

In [5]: a = 3.14159
        b = 5
        c = a * (b**2)

In [6]: x = 2
        print x
        2

In [7]: print 'x = ', x
        x =  2

In [8]: x, y = 2, 3
        print 'x = ', x
        print 'y = ', y
        x =  2
        y =  3
```

Variables

- Uno de los usos habituales de las sentencia de asignación consiste en realizar una actualización sobre una variable – en la cual el valor nuevo de esa variable depende del antiguo: x = x+1

- Esto quiere decir "'toma el valor actual de x, añadele 1, y luego actualiza x con el nuevo valor".

Variables

- En el siguiente ejemplo se intercambian los valores dos variables x e y

```
In [9]: x, y = 2, 3
        x, y = y, x
        print 'x = ', x
        print 'y = ', y

        x =  3
        y =  2
```

Variables

- El mismo ejemplo que antes, pero sin hacer doble asignación

```
In [1]: x = 2
        y = 3
        z = x
        x = y
        y = z
        print 'x = ', x
        print 'y = ', y

x =  3
y =  2
```

Variables

- Observar:
  - Si se intenta actualizar una variable que no existe, se obtiene un error, ya que Python evalúa el lado derecho antes de asignar el valor a x.
  - Antes de poder actualizar una variable, se debe inicializar mediante una asignación. A continuación se puede actualizar la variable aumentándola(incrementar) o disminuyendo(decrementar)

```
In [13]: x=0
         x=x+1
```

Palabras reservadas en Python

- Python reserva 31 palabras claves para su propio uso:

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

Operadores

- Los operadores son símbolos especiales que representan cálculos, como la suma o la multiplicación. Los valores a los cuales se aplican esos operadores reciben el nombre de operandos.

Operadores

- Los principales operadores sobre los tipos int y float son:
  - i+j suma
  - i-j resta
  - i*j multiplicación
  - i/j división de dos números. Si son enteros, el resultado es un entero, y si son reales, el resultado es un real.
  - i//j cociente de la división entera
  - i%j resto de la división entera
  - i**j i elevado a la potencia j
  - i==j i igual que j
  - i!=j i distinto que j
  - i>j i mayor que j, y de forma similar: >=, <, <=

Operadores

- En el operador de la división, cuando ambos operandos son enteros, el resultado es también un entero. Sin embargo si cualquiera de los operandos es un número en punto flotante, entonces realiza división en punto flotante, y el resultado es un float.

```
In [10]: minuto=59

In [11]: minuto/60
Out[11]: 0

In [12]: minuto/60.0
Out[12]: 0.9833333333333333
```

Operadores

- Se pueden usar los operadores con las cadenas

```
In [9]: 3*'a'
Out[9]: 'aaa'

In [10]: 'a'+'a'
Out[10]: 'aa'
```

Operadores

- Pero existen algunas particularidades cuando se usan los operadores sobre las cadenas.

```
In [11]: 'a'*'a'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-19dbecaed6ac> in <module>()
----> 1 'a'*'a'

TypeError: can't multiply sequence by non-int of type 'str'

In [12]: '4' < 3
Out[12]: False

In [13]: '2' < 3
Out[13]: False
```

Operadores

```
In [3]: len('Neurociencia')
Out[3]: 12

In [2]: 'Neurociencia'[5]
Out[2]: 's'

In [4]: 'Neurociencia'[12]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-4-117c58572826> in <module>()
----> 1 'Neuroscience'[12]

IndexError: string index out of range

In [6]: 'Neurociencia'[0:5]
Out[6]: 'Neuro'

In [1]: 'Neurociencia'[5:len('Neurociencia')]
Out[1]: 'ciencia'
```

Expresiones

- Una expresión es una combinación de valores, variables y operadores. Un valor por si mismo se considera una expresión, y también lo es una variable.
- Las expresiones tienen un tipo. Así por ejemplo:

  6 + 7 es una expresión que representa un entero

Expresiones

- Cuando en una expresión aparece más de un operador, el orden de evaluación depende de las reglas de precedencia. Para los operadores matemáticos, Python sigue las convenciones matemáticas:
  - El orden de los operadores es: paréntesis, exponenciales, multiplicación/división, suma/resta.
  - Cuando existe la misma precedencia, se evalúa de izquierda a derecha.

Expresiones

```
In [4]: 4**1 + 1
Out[4]: 5

In [5]: 6*1**2
Out[5]: 6

In [6]: (-1)**3*3
Out[6]: -3

In [7]: 6*3/2
Out[7]: 9

In [8]: 3/2*6
Out[8]: 6
```

Comentarios

- En Python comienzan con el símbolo #, de forma que todo lo que va desde # hasta el final de la línea es ignorado y no afecta para al programa.

```
In [14]: #Calcula el porcentaje de hora transcurrido
         porcentaje = (minuto*100)/60
```

Comentarios

- En el ejemplo anterior el comentario aparece como una línea completa, pero también puede ponerse comentarios al final de una línea

```
In [15]: porcentaje = (minuto * 100) / 60 # porcentaje de una hora
```

Entrada de información

- Python proporciona una función llamada raw_input que recibe la entrada desde el teclado, de forma que cuando se llama el programa se detiene y espera a que el usuario escriba algo. Cuando el usuario pulsa Intro, el programa continúa y la función devuelve como una cadena aquello que el usuario escribió.

```
In [1]: entrada = raw_input()
        Hola

In [2]: print entrada
        Hola
```

Entrada de Información

- Antes de recibir cualquier dato desde el usuario, es mejor escribir un mensaje explicando qué debe introducir. Se puede pasar una cadena a raw_input, que será mostrada al usuario antes de que el programa se detenga para recibir su entrada:

```
In [1]: nombre = raw_input('¿Cómo te llamas?\n')

Â¿CÃ³mo te llamas?
Juan

In [2]: print nombre

Juan
```

Entrada de información

- La secuencia \n al final del mensaje representa un newline, que es un carácter especial que provoca un salto de línea. Por eso la entrada del usuario aparece debajo del mensaje.

Entrada de información

- Si es espera que el usuario escriba un entero, puedes intentar convertir el valor de retorno a int usando la función int(), pero si el usuario escribe algo que no sea una cadena de dígitos, obtiene un error:

```
In [3]: velocidad = raw_input('Velocidad inicial\n')

        Velocidad inicial
        30

In [4]: int(velocidad)

Out[4]: 30
```

Expresiones booleanas

- Una expresión booleana es aquella que puede ser verdadera (True) o falsa (False).
- True y False son valores especiales que pertenecen al tipo bool (booleano)

```
In [5]: type(True)
Out[5]: bool

In [6]: type(False)
Out[6]: bool
```

Expresiones booleanas

- Los ejemplos siguientes usan el operador ==, que compara dos operandos y devuelve True si son iguales y False en caso contrario:

```
In [7]: 5==5

Out[7]: True

In [8]: 5==6

Out[8]: False
```

Expresiones booleanas

- Los principales operadores de comparación son:
  - x == y # x es igual que y
  - x != y # x es distinto de y
  - x > y # x es mayor que y
  - x < y # x es menor que y
  - x >= y # x es mayor o igual que y
  - x <= y # x es menor o igual que y
  - x is y # x es lo mismo que y
  - x is not y # x no es lo mismo que y

Operadores lógicos

- Existen tres operadores lógicos que se usan en las expresiones condicionales:
  - not representa la negación
  - and cierto si las dos expresiones que relaciona son ciertas, y falso en caso contrario.
  - or falso si las dos expresiones que relaciona son falsas, y cierto en caso contrario.

Operadores lógicos

- Por ejemplo:
  - x >0 and x <10 es verdadero sólo cuando x es mayor que 0 *y* menor que 10.
  - n%2 == 0 or n%3 == 0 es verdadero si el número es divisible por 2 *o* por 3.
  - not (x>y) es verdadero si x es menor o igual que y.
- Observar que cualquier número distinto de cero se interpreta como “verdadero.”

```
In [9]: 17 and True
Out[9]: True
```

Evaluación en cortocircuito

- Cuando Python detecta que no se gana nada evaluando el resto de una expresión lógica, detiene su evaluación y no realiza el cálculo del resto de la expresión.
- Cuando la evaluación de una expresión lógica se detiene debido a que ya se conoce el valor final, eso es conocido como cortocircuitar la evaluación.

Condicionales

- Las expresiones condicionales facilitan la codificación de estructuras que bifurcan la ejecución del código en varias ramas o caminos de ejecución.
- La estructura de selección más simple es:

  if expresión booleana:

  ejecutar código1

```
In [12]: x = 3
         if x > 0:
             print "x es positivo"

x es positivo
```

Condicionales

Observar:

- La expresión booleana después de la sentencia if recibe el nombre de condición. La sentencia if se finaliza con un carácter de dos-puntos (:) y la(s) línea(s) que van detrás de la sentencia if van indentadas. Este código se denomina bloque.
- Si la condición lógica es verdadera, la sentencia indentada será ejecutada. Si la condición es falsa, la sentencia indentada será omitida.
- No hay límite en el número de sentencias que pueden aparecer en el cuerpo, pero debe haber al menos una. A veces puede resultar útil tener un cuerpo sin sentencias, usándose en este caso la sentencia pass, que no hace nada.

Condicionales

- La segunda forma de la sentencia if es la ejecución alternativa, en la cual existen dos posibilidades y la condición determina cual de ellas sería ejecutada:

  if expresión booleana:

    ejecutar código1

  else:

    ejecutar código2

```
In [11]: x = -5
         if x > 0:
             print "x es positivo"
         else:
             print "x es negativo"

x es negativo
```

```
In [10]: x = 14
         if (x > 0) and (x <= 20):
             print "x es positivo"
             print "x es menor o igual que 20"
         else:
             print "x es negativo o bien muy grande"

x es positivo
x es menor o igual que 20
```

Condicionales

- Dado que la condición debe ser obligatoriamente verdadera o falsa, solamente una de las alternativas será ejecutada. Las alternativas reciben el nombre de ramas, dado que se trata de ramificaciones en el flujo de la ejecución.

Condicionales

- La tercera forma de la sentencia if es el condicional encadenado que permite que haya más de dos posibilidades o ramas:

```
if expresion booleana:
        ejecutar codigo 1
elif:
        ejecutar codigo 2
else:
        ejecutar código por defecto
```

```
In [1]: x = 1
        y = 2
        z = 3

        if x < y and x < z:
            print 'x es el más pequeño'
        elif y < z:
            print 'y es el más pequeño'
        else:
            print 'z es el más pequeño'

x es el más pequeño
```

Condicionales

- Observar:
  - No hay un límite para el número de sentencias elif. Si hay una clausula else, debe ir al final, pero tampoco es obligatorio que ésta exista.
  - Cada condición es comprobada en orden. Si la primera es falsa, se comprueba la siguiente y así con las demás. Si una de ellas es verdadera, se ejecuta la rama correspondiente, y la sentencia termina. Incluso si hay más de una condición que sea verdadera, sólo se ejecuta la primera que se encuentra.

Condicionales

- Un condicional puede también estar anidado dentro de otro:

```
In [ ]: x = 14
if (x > 0):
    print "x es positivo"
    if (x <= 20):
        print "x es menor o igual que 20"
    else:
        print "x es más grande que 20"
else:
    print "x es negativo"
```

Condicionales

- El condicional exterior contiene dos ramas. La primera contiene otra sentencia if, que tiene a su vez sus propias dos ramas. Esas dos ramas son ambas sentencias simples, pero podrían haber sido sentencias condicionales también. La segunda rama ejecuta una sentencia simple.

Condicionales

- Los condicionales anidados pueden volverse difíciles de leer por lo que deben evitarse y usar operadores lógicos que permiten simplificar las sentencias condicionales anidadas.

Excepciones

- Las llamadas “try / except” permiten añadir ciertas sentencias para que sean ejecutadas en caso de que cierta secuencia de instrucciones generen un error. Las sentencias extras serán ignoradas si no se produce ningún error.
- Python comienza ejecutando la secuencia de sentencias del bloque try. Si todo va bien, se saltará todo el bloque except y terminará. Si ocurre una excepción dentro del bloque try, Python saltará fuera de ese bloque y ejecutará la secuencia de sentencias del bloque except.

Excepciones

```
In [12]: ent = raw_input('Introduzca la Temperatura Fahrenheit:')
         try:
             fahr = float(ent)
             cel = (fahr - 32.0) * 5.0 / 9.0
             print cel
         except:
             print 'Por favor, introduzca un número'

         Introduzca la Temperatura Fahrenheit:
         Por favor, introduzca un número
```

Excepciones

- Gestionar una excepción con una sentencia try recibe el nombre de capturar una excepción. En este ejemplo, la clausula except muestra un mensaje de error.
- En general, capturar una excepción da la oportunidad de corregir el problema, volverlo a intentar o terminar el programa con elegancia.

Bucles

- Los bucles permiten la repetición y generalmente se construyen así:
  - Se inicializan una o más variables antes de que el bucle comience
  - Se realiza alguna operación con cada elemento en el cuerpo del bucle, posiblemente cambiando las variables dentro de ese cuerpo.
  - Se revisan las variables resultantes cuando el bucle se completa

Bucles

- El primer tipo de bucle es el "while", que tiene la siguiente estructura:

  while (expresion booleana):

    codigo (hay que asegurarse que la expresión booleana cambia en este trozo de código)

Bucles

In [1]:
```
n = 5
while n > 0:
    print n
    n = n-1
print "¡Fin!"
```

```
5
4
3
2
1
¡Fin!
```

In [1]:
```
x = 3
ans = 0 #Inicializa esta variable a cero
bucle = x
while (bucle != 0):
    ans = ans + x
    bucle = bucle - 1
print str(x) + '*' + str(x) + ' = ' + str(ans)
```

```
3*3 = 9
```

Bucles

- Observar que:
  - *El bucle nunca se ejecuta cuando x=0.*
  - *El bucle nunca terminará si empieza con x<0.*

Bucles

- Cada vez que se ejecuta el cuerpo del bucle se dice que se realiza una iteración.
- El cuerpo del bucle debe cambiar el valor de una o más variables, de modo que la condición pueda en algún momento evaluarse como falsa y el bucle termine. La variable que cambia cada vez que el bucle se ejecuta y controla cuándo termina éste, recibe el nombre de variable de iteración.

Bucles

- Si no hay variable de iteración, el bucle se repetirá para siempre, resultando así un bucle infinito.
- A veces no se sabe si hay que terminar un bucle hasta que se ha recorrido la mitad del cuerpo del mismo. En ese caso se puede crear un bucle infinito a propósito y usar la sentencia break para salir explícitamente cuando se haya alcanzado la condición de salida.

Bucles

```
In [3]: while True:
            linea = raw_input('> ')
            if linea == 'fin':
                    break
            print linea
        print '¡Terminado!'

> fin
¡Terminado!
```

Bucles

- La condición del bucle es True, lo cual es verdadero siempre, así que el bucle se repetirá hasta que se ejecute la sentencia break. Cada vez que se entre en el bucle, se pedirá una entrada al usuario. Si el usuario escribe fin, la sentencia break hará que se salga del bucle. En cualquier otro caso, el programa repetirá cualquier cosa que el usuario escriba y volverá al principio del bucle

Bucles

- Algunas veces, estando dentro de un bucle se necesita terminar con la iteración actual y saltar a la siguiente de forma inmediata. En ese caso se puede utilizar la sentencia continue para pasar a la siguiente iteración sin terminar la ejecución del cuerpo del bucle para la actual.

Bucles

```
In [4]: while True:
            linea = raw_input('> ')
            if linea[0] == '#' :
                continue
            if linea == 'fin':
                break
            print linea
        print '¡Terminado!'
```

```
> hola
hola
> #hola
> adios
adios
> fin
¡Terminado!
```

Bucles

- Todas las líneas se imprimen en pantalla, excepto la que comienza con el símbolo de almohadilla, ya que en ese caso se ejecuta continue, finaliza la iteración actual y salta de vuelta a la sentencia while para comenzar la siguiente iteración, de modo que se omite la sentencia print.

Bucles

- El siguiente tipo de bucle es el for, que tiene la siguiente estructura:

```
for variable in secuencia:
    código
```

- El bucle for se repite a través de un conjunto conocido de elementos, de modo que ejecuta tantas iteraciones como elementos hay en el conjunto.

Bucles

- Es útil utilizar la función 'range' para crear una secuencia. Range puede tomar uno o dos valores:
  - Si toma dos valores, genera todos los enteros desde la primer entrada hasta la segunda entrada-1.Por ejemplo: range(2, 5) = (2, 3, 4).
  - Y si toma un sólo parámetro, entonces range(x) = range(0,x)

Bucles

```
In [2]: x = 5
        for i in range(0,x):
            print i
```

```
0
1
2
3
4
```

```
In [9]: x = 5
        for i in range(x):
            print i
            x = 10
```

```
0
1
2
3
4
```

Bucles

- Los bucles pueden estar anidados

```
In [5]: x = 4
        y = 3
        for j in range(y):
            for i in range(x):
                print str(i) + " " + str(j)
```

```
0 0
1 0
2 0
3 0
0 1
1 1
2 1
3 1
0 2
1 2
2 2
3 2
```

```
In [10]: x = 4
         for j in range(x):
             for i in range(x):
                 print i
                 x = 2
```

```
0
1
2
3
0
1
0
1
0
1
```

Bucles

- En el siguiente ejemplo permite al usuario adivinar el número generado por el ordenador entre 1-10.

In [ ]:
```
#Importamos algunas librerias de python
import random, math

respuesta = random.random()
#La función random.random devuelve un número real tal que 0 <= numero < 1, pero como queremos que esté entre 1-10, entonces

respuesta = math.ceil(respuesta * 10)
#La función math.ceil nos da el número redondeado al siguiente entero.

intentos = 1 #Se inicializa la variable a 1
entrada = int(raw_input('Adivina un número entre 1 - 10: ')) #Se obtiene la entrada del usuario

while entrada!= respuesta:
    if entrada > respuesta:
        print('Es demasiado grande')
    else:
        print ('Es demasiado pequeño')
    entrada = int(raw_input('Adivina un número entre 1 - 10: '))
    intentos = intentos + 1

#Escribe la salida
print 'Lo conseguiste. La respuesta era ' + str(respuesta)
print 'Necesitaste ' + str(intentos) + ' intentos.'
```

Funciones

- Una función es una secuencia de sentencias que realizan una operación y que reciben un nombre. Cuando se define una función, se especifica el nombre y la secuencia de sentencias, y se puede "llamar" a la función por ese nombre.

Funciones

- Junto al nombre de la función aparece una expresión entre paréntesis que recibe el nombre de argumento de la función. El argumento es un valor o variable que se pasa a la función como parámetro de entrada. El resultado de la función type es el tipo del argumento.
- Es habitual decir que una función “toma” (o recibe) un argumento y “retorna” (o devuelve) un resultado. El resultado se llama valor de retorno.

Funciones

- Python proporciona un número importante de funciones internas, que pueden ser usadas sin necesidad de tener que definirlas previamente.

Funciones

- Por ejemplo:
  - Las funciones max y min dan respectivamente el valor mayor y menor de una lista:

```
In [5]: max('¡Hola, mundo!')
Out[5]: '\xc2'

In [6]: min('¡Hola, mundo!')
Out[6]: ' '
```

  - La función len devuelve cuántos elementos hay en su argumento. Si el argumento es una cadena devuelve el número de caracteres que hay en la cadena.

```
In [7]: len('Hola, mundo')
Out[7]: 11
```

Funciones

- Observar:
  - Estas funciones pueden operar con cualquier conjunto de valores.
  - Se deben tratar los nombres de las funciones internas como si fueran palabras reservadas.

Funciones

- Python también proporciona funciones internas que convierten valores de un tipo a otro:
  - int toma cualquier valor y lo convierte en un entero, si puede. Convierte valores en punto flotante a enteros descartando la parte decimal:
  - float convierte enteros y cadenas en números de punto flotante
  - str convierte su argumento en una cadena.

Funciones

```
In [8]: int('32')
Out[8]: 32

In [9]: int(3.99999)
Out[9]: 3

In [10]: float(32)
Out[10]: 32.0

In [11]: str(3.14159)
Out[11]: '3.14159'
```

Funciones

- El módulo random proporciona funciones que generan números pseudoaleatorios. Antes de usarlo hay que importarlo y para acceder a sus funciones es necesario especificar el nombre del módulo y el nombre de la función separados por un punto.

Funciones

- La función random devuelve un número flotante aleatorio entre 0.0 y 1.0 (incluyendo 0.0, pero no 1.0). Cada vez que se llama a random, se obtiene el número siguiente de una larga serie.

```
In [13]: import random
         for i in range(4):
             x = random.random()
             print x
```

```
0.175387431517
0.33338879632
0.693998673603
0.642592146423
```

Funciones

- La función randint toma los parámetros inferior y superior, y devuelve un entero entre inferior y superior (incluyendo ambos extremos).

```
In [14]: random.randint(5, 10)
Out[14]: 9
```

- Para elegir un elemento de una secuencia aleatoriamente, se puede usar choice.

```
In [15]: t = [1, 2, 3]
         random.choice(t)
Out[15]: 1
```

- El módulo random también proporciona funciones para generar valores aleatorios de distribuciones continuas.

Funciones

- Python tiene un módulo matemático (math), que proporciona la mayoría de las funciones matemáticas habituales.
- Algunos ejemplos:
  - La función log10 cálcula el logaritmo base 10 y la función log calcula logaritmos en base e.
  - Las funciones trigonométricas sin cos, tan,… toman argumentos en radianes.
  - La expresión math.pi toma una aproximación del valor de pi.

Funciones

```
In [17]: import math

In [19]: decibelios = 10 * math.log10(23)
         print decibelios

         13.6172783602

In [20]: radianes = 0.7
         altura = math.sin(radianes)
         print altura

         0.644217687238

In [21]: grados = 45
         radianes = grados / 360.0 * 2 * math.pi
         math.sin(radianes)

Out[21]: 0.7071067811865475
```

Funciones

- Una definición de función especifica el nombre de una función nueva y la secuencia de sentencias que se ejecutan cuando esa función es llamada. Una vez definida una función, se puede reutilizar una y otra vez a lo largo de todo el programa.

Funciones

- Para crear una función se utiliza la palabra reservada “def”:

```
In [3]: def imprimir_hola():
            print "hola"
```

Funciones

- A continuación de la palabra def aparece el nombre del la función (en el ejemplo imprimir_hola) y a continuación aparecen unos paréntesis que están reservados para los parámetros, y finaliza con “:” . Esta línea se denomina **cabecera** de la función

```
In [5]: def imprimir_hola(nombre):
            print "Hola " + nombre + "!"
```

Funciones

- A continuación de los paréntesis, aparece el código que se ejecuta cuando se llama a la función. Este trozo de código, se denomina **cuerpo de la función** y debe estar indentado. El cuerpo puede contener cualquier número de sentencias.

Funciones

- Las reglas para los nombres de las funciones son los mismos que para las variables: se pueden usar letras, números y algunos signos de puntuación, pero el primer carácter no puede ser un número. No se puede usar una palabra clave como nombre de una función, y se debería evitar también tener una variable y una función con el mismo nombre.

Funciones

- Las funciones con paréntesis vacíos después del nombre indican que esta función no toma ningún argumento.

Funciones

- Si escribes una definición de función en modo interactivo, el intérprete muestra puntos suspensivos (...) para informar de que la definición no está completa. Para finalizar la función, se debes introducir una línea vacía (esto no es necesario en un script).

Funciones

- Al definir una función se crea una variable con el mismo nombre cuyo valor tiene tipo “function” de forma que cuando se introduce únicamente el nombre de la función sin paréntesis, en vez de ejecutarse, se imprime información de la función. En este caso indica que se trata de un objeto función denominado "imprimir_hola".

```
In [2]: imprimir_hola

Out[2]: <function __main__.imprimir_hola>
```

Funciones

- La sintaxis para llamar a una función definida es la misma que para las funciones internas.

```
In [4]: imprimir_hola()
hola
```

- Una vez que se ha definido una función puede usarse dentro de otra.
- La definición de la función debe ser ejecutada antes de que la función se llame por primera vez.

Funciones

- Las definiciones de funciones son ejecutadas exactamente igual que cualquier otra sentencia, pero su resultado consiste en crear objetos del tipo función. Las sentencias dentro de cada función son ejecutadas solamente cuando se llama a esa función, y la definición de una función no genera ninguna salida.

Funciones

- En Python no se especifica el tipo de parámetro, sólo los nombres de las funciones ni tampoco se especifica lo que la función retorna, sino que se hace en el cuerpo de la función. En el ejemplo anterior, no se retorna nada explícitamente, por lo que implícitamente retorna el valor None.

```
In [6]: imprimir_hola("Juan")

        Hola Juan!

In [7]: imprimir_hola("Maria")

        Hola Maria!
```

Funciones

- En la siguiente función se calcula la mediana usando el método sort para ordenar una lista:

```
In [8]: def mediana(lista):
            lista.sort()
            if (len(lista)%2):
                return lista[len(lista)/2]
            else:
                return ((lista[len(lista)/2-1]+lista[len(lista)/2])/2.0)

In [9]: mediana([3,0,6,2,10])
Out[9]: 3

In [10]: mediana([1,3,4,7])
Out[10]: 3.5
```

```
In [12]: valores = [7,3,5,4,1]
         mediana(valores)
Out[12]: 4

In [13]: valores
Out[13]: [1, 3, 4, 5, 7]
```

Funciones

- Se llama flujo de ejecución al orden en que las sentencias son ejecutadas dentro de un programa. La ejecución siempre comienza en la primera sentencia del programa, y las sentencias son ejecutadas una por una en orden de arriba hacia abajo. Las *definiciones* de funciones no alteran el flujo de la ejecución del programa debido a que las sentencias dentro de una función no son ejecutadas hasta que se llama a esa función.

Funciones

- Una llamada a una función es como un desvío en el flujo de la ejecución. En vez de pasar a la siguiente sentencia, el flujo salta al cuerpo de la función, ejecuta todas las sentencias que hay allí, y después vuelve al punto donde lo dejó.

Funciones

- Las funciones que disponen de argumentos, éstos son asignados a variables llamadas parámetros. En este sentido las mismas reglas de composición que se aplican a las funciones internas, también se aplican a las funciones definidas por el usuario, de modo que podemos usar cualquier tipo de expresión como argumento, la cual será evaluada antes de que la función sea llamada.

Funciones

- El nombre de la variable que se pasa como argumento no tiene nada que ver con el nombre del parámetro de manea que dentro de la función recibirá el nombre del parámetro.

Funciones

- Es muy importante saber si en las funciones se cambian las variables que son pasadas como parámetros. Esto se denomina "efecto lateral", y dependiendo de cada caso, puede ser necesario o no.

Funciones

- Para especificar una función con más de un argumento, basta especificar los argumentos dentro de () en la definición de la función. Cuando se definen los argumentos, éstos pueden tener valores por defectos.

```
In [14]: def imprimir_hola(nombre, condicion = False):
             str = "Hola " + nombre + "!"
             if (condicion):
                 print str.upper()
             else:
                 print str

In [22]: imprimir_hola("Mama")

         Hola Mama!

In [16]: imprimir_hola(condicion=True, nombre="Mama")

         HOLA MAMA!
```

Funciones

- Desde una función puede llamarse a otras funciones, facilitando de esta manera la descomposición de un problema, y resolverlo mediante una combinación de llamadas a funciones. En el ejemplo siguiente se crean dos funciones, una que crea un mensaje, y otra que usa la anterior para imprimirlo.

```
In [19]: def crear_saludo(nombre):
             return "Hola, {0}".format(nombre)

         def imprimir_saludo(nombre):
             print crear_saludo(nombre)

In [21]: imprimir_saludo("Tom")

         Hola, Tom
```

Funciones

- Python facilita la documentación de las funciones. Basta incluir una cadena entre la línea dónde se define def y el cuerpo usando una tripleta de comillas """:

```
In [17]: def mediana(lista):
             """Encuentra la mediana de una lista de números y devuelve el valor.
             Para listas con un número impar de elementos, se trata del valor medio de la lista.
             Para listas con un número par de elementos, se trata de la media de las valores medios.
             Efectos laterales: la lista es ordenada en esta función"""
             lista.sort()
             if (len(lista)%2):
                 return lista[len(lista)/2]
             else:
                 return ((lista[len(lista)/2-1]+lista[len(lista)/2])/2.0)

In [18]: help(mediana)
         Help on function mediana in module __main__:

         mediana(lista)
             Encuentra la mediana de una lista de números y devuelve el valor.
             Para listas con un número impar de elementos, se trata del valor medio de la lista.
             Para listas con un número par de elementos, se trata de la media de las valores medios.
             Efectos laterales: la lista es ordenada en esta función
```

Funciones

- Hay dos tipos de funciones:
  - Productivas son aquellas que producen resultados.
  - Estériles son aquellas que realizan alguna acción pero no devuelven un valor.
- Cuando se llama a una función productiva se querrá hacer algo con el resultado, para lo cual hay que asignarlo a una variable o usarlo como parte de una expresión

Funciones

- Las funciones estériles pueden mostrar algo en la pantalla o tener cualquier otro efecto, pero no devuelven un valor. Si se intenta asignar el resultado a una variable, se obtiene un valor especial llamado None (nada) que tiene su propio tipo:

```
In [1]: print type(None)
<type 'NoneType'>
```

Funciones

- Para devolver un resultado desde una función se usa la sentencia return dentro de ella.

```
In [2]: def sumados(a, b):
            suma = a + b
            return suma
        x = sumados(3, 5)
        print x
```

```
8
```

Funciones

- Cuando se ejecuta el programa anterior, la sentencia print muestra “8”, ya que la función ha sido llamada con 3 y 5 como argumentos. Dentro de la función, los parámetros a y b equivalen a 3 y a 5 respectivamente, se calcula la suma de ambos y se guarda en una variable local a la función llamada suma. Se usa la sentencia return para enviar el valor calculado de vuelta al código de llamada como resultado de la función, que es asignado a la variable x y se muestra en pantalla.

Cadenas

- Una cadena es una secuencia de caracteres. Para acceder a los caracteres de uno en uno se usa el operador corchete:

```
In [4]: fruta = 'banana'
        letra = fruta[1]
        print letra

a
```

Cadenas

- La expresión entre corchetes recibe el nombre de índice, e indica a qué carácter de la secuencia se desea acceder. El índice siempre comienza en 0.
- Se pueden usar índices negativos, que cuentan hacia atrás desde el final de la cadena.
- Se puede utilizar cualquier expresión, incluyendo variables y operadores, como índice, pero el valor del índice siempre debe ser un entero.

Cadenas

- len es una función interna que devuelve el número de caracteres de una cadena.

```
In [5]: fruta = 'banana'
        len(fruta)

Out[5]: 6
```

Cadenas

- Algunas operaciones implican procesar una cadena carácter por carácter. A menudo se empieza por el principio, se van seleccionando caracteres de uno en uno, se hace algo con ellos, y se continúa hasta el final. Este modelo de procesado recibe el nombre de recorrido y se puede hacer con un bucle for.

```
In [ ]: for car in fruta:
            print car
```

- Cada vez que se recorre el bucle, el carácter siguiente de la cadena es asignado a la variable car. El bucle continúa hasta que no quedan caracteres.

Cadenas

- Un segmento de una cadena recibe el nombre de rebanada (slice), y se puede seleccionar con el operador [n:m] que devuelve la parte de la cadena desde el “n-´esimo” carácter hasta el “m-´esimo”, incluyendo el primero pero excluyendo el último.

```
In [6]: s = 'Monty Python'
        print s[0:5]

Monty
```

Cadenas

- Si se omite el primer índice la rebanada comenzará al principio de la cadena, y si se omite es el segundo, la rebanada abarcará hasta el final de la cadena.

```
In [7]: fruta = 'banana'
        fruta[:3]

Out[7]: 'ban'

In [8]: fruta[3:]

Out[8]: 'ana'
```

Cadenas

- Si el primer índice es mayor o igual que el segundo, el resultado será una cadena vacía, representada por dos comillas. Una cadena vacía no contiene caracteres y tiene una longitud 0.

```
In [9]: fruta = 'banana'
        fruta[3:3]

Out[9]: ''
```

Cadenas

- Las cadenas son inmutables, lo cual significa que no se puede cambiar una cadena existente.

```
In [10]: saludo = '¡Hola, mundo!'
         saludo[0] = 'J'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-7600ef133cf0> in <module>()
      1 saludo = '¡Hola, mundo!'
----> 2 saludo[0] = 'J'

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

Cadenas

- El operador in es un operador booleano que toma dos cadenas y devuelve True (verdadero) si la primera aparece como subcadena dentro de la segunda.

```
In [11]: 'a' in 'banana'

Out[11]: True
```

Cadenas

- Los operadores de comparación funcionan con las cadenas, sin embargo hay que tener en cuenta que en Python que las letras mayúsculas van antes que las minúsculas. En esté sentido un método para evitar este problema es convertir las cadenas a minúsculas o mayúsculas antes de realizar la comparación.

Cadenas

- Las cadenas son objetos en Python y tiene asociados un conjunto de métodos o funciones aplicables a cualquier cadena.
- La función type muestra el tipo de cualquier objeto y la función dir muestra los métodos disponibles.

```
In [11]: 'a' in 'banana'

Out[11]: True

In [12]: cosa = '¡Hola, mundo!'
         type(cosa)

Out[12]: str

In [13]: dir(cosa)

Out[13]: ['__add__',
          '__class__',
          '__contains__',
          '__delattr__',
          '__doc__',
```

Cadenas

- La función help permite obtener información sobre cada método.

```
In [14]: help(str.capitalize)

Help on method_descriptor:

capitalize(...)
    S.capitalize() -> string

    Return a copy of the string S with only its first character
    capitalized.
```

Cadenas

- Llamar a un método es similar a llamar a una función, toma argumentos y devuelve un valor, pero la sintaxis es diferente. Sin embargo un método se usa uniendo el nombre del método al de la variable, utilizando el punto como delimitador. Por ejemplo el el método upper toma una cadena y devuelve otra nueva con todas las letras en mayúsculas:

```
In [15]: palabra = 'banana'
         nueva_palabra = palabra.upper()
         print nueva_palabra

         BANANA
```

Cadenas

- Esta forma de notación con punto especifica el nombre del método, upper, y el nombre de la cadena a la cual se debe aplicar ese método, palabra. Los paréntesis vacíos indican que el método no toma argumentos.
- Una llamada a un método se denomina invocación.

Cadenas

- Otros métodos:
  - find busca la posición de una cadena dentro de otra. También puede tomar un segundo argumento que indica en qué posición debe comenzar la búsqueda.

```
In [16]: palabra = 'banana'
         indice = palabra.find('a')
         print indice

         1

In [18]: palabra.find('na', 3)

Out[18]: 4
```

Cadenas

- El método strip elimina espacios en blanco del principio y del final de una cadena.

```
In [20]: linea = ' Y allá vamos '
         linea.strip()

Out[20]: 'Y all\xc3\xa1 vamos'
```

- startswith comprueba si una cadena comienza con la subcadena dada como parámetro, distinguiendo entre mayúsculas y minúsculas.

```
In [22]: linea = 'Que tengas un buen día'
         linea.startswith('Que')

Out[22]: True
```

Cadenas

- El método lower convierte a minúsculas una cadena dada.

```
In [23]: linea = 'Que tengas un buen día'
         linea.lower()

Out[23]: 'que tengas un buen d\xc3\xada'
```

- El método rstrip retira los espacios en blanco de la parte derecha de una cadena.

```
In [4]: cadena='coche     '
        cadena.rstrip()

Out[4]: 'coche'
```

- Se pueden hacer múltiples llamadas a un método en una única expresión.

Cadenas

- El operador de formato % permite construir cadenas reemplazando parte de esas cadenas con los datos almacenados en variables. Observar que cuando se aplica a enteros % es el operador módulo pero cuando el primer operando es una cadena % es el operador formato.

Cadenas

- El primer operando es la cadena a formatear, que contiene una o más secuencias de formato, que especifican cómo será formateado el segundo operador. El resultado es una cadena.

```
In [24]: camellos = 42
         '%d' % camellos

Out[24]: '42'
```

- La secuencia de formato '%d' quiere decir que el segundo operador debe ser formateado como un entero (d indica “decimal”).El resultado es la cadena '42‘.

Cadenas

- Una secuencia de formato puede aparecer en cualquier sitio de la cadena, de modo que puedes insertar un valor en una frase.

```
In [25]: camellos = 42
         'He divisado %d camellos.' % camellos

Out[25]: 'He divisado 42 camellos.'
```

Cadenas

- Si hay más de una secuencia de formato en la cadena, el segundo argumento debe ser una tupla. Cada secuencia de formato se corresponde con un elemento de la tupla, en orden.

```
In [26]: camellos = 42
         'En %d años he divisado %g %s.' % (3, 0.1, 'camellos')
         'En 3 a˜nos he divisado 0.1 camellos.'

Out[26]: 'En 3 a\xcb\x9cnos he divisado 0.1 camellos.'
```

Cadenas

- El número de elementos en la tupla debe coincidir con el número de secuencias de formato en la cadena. El tipo de los elementos debe coincidir también con las secuencias de formato.

```
In [27]: '%d %d %d' % (1, 2)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-76a835ca51c7> in <module>()
----> 1 '%d %d %d' % (1, 2)

TypeError: not enough arguments for format string

In [28]: '%d' % 'dólares'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-28-539aa4f2616b> in <module>()
----> 1 '%d' % 'dólares'

TypeError: %d format: a number is required, not str
```

Cadenas

- En el primer ejemplo, no hay suficientes elementos y en el segundo el elemento es de tipo incorrecto.