

CursoPython (/github/LlanosTobarra/CursoPython/tree/master)

/ Sesión 1 (/github/LlanosTobarra/CursoPython/tree/master/Sesión 1)

Introducción a Python

Comentarios

Es frecuente que conforme programemos queramos añadir comentarios para documentar nuestro código. En Python los comentarios sencillos incluidos en cualquier parte del código comienzan con #.

Si vamos a realizar un comentario más largo, que se extienda por varias líneas, usamos tres comillas simples para iniciar el comentario y lo cerramos con tres comillas simples. Estos comentarios se utilizan para documentar el código de forma "oficial". Veremos su utilidad más adelante.

Todos los comentarios son ignorados a la hora de compilar el programa.

```
In [17]: #comentario sencillo
```

Variables

Las variables son zonas de memoria donde vamos a almacenar nuestros datos temporalmente durante la ejecución del programa. Como ya hemos visto en Python las variables se definen conforme son necesarias, sin necesidad de una definición formal donde se indique el tipo de datos de antemano. Python se encarga de determinar el tipo de datos de la variable en tiempo de ejecución. Si queremos saber el tipo de una variable podemos usar la función **type()**. Por ejemplo

```
In [9]: a=2
        b=a+3
        print a,b
```

```
2 5
```

```
In [10]: type(a)
```

```
Out[10]: int
```

```
In [11]: a="Hola mundo,"
        b=" ¿que tal?"
        print a+b
```

```
Hola mundo, ¿que tal?
```

```
In [12]: type(a)
```

```
Out[12]: str
```

Como hemos visto el tipo de datos de `a` y `b` ha cambiado a lo largo de la ejecución de las celdas, según ha cambiado el valor que le hemos asignado. Hemos de tener cuidado con esta característica de Python, ya que los errores derivados de los tipos de datos aparecen en el momento de la ejecución, no en tiempo de compilación. Por ejemplo:

```
In [13]: a+3
```

```
Out[13]: 15
```

Además en Python no tenemos mecanismos para crear constantes (variables cuyo valor una vez definido no varía a lo largo del programa). Por convención general, cuando queremos definir una variable se suele crear con el nombre en mayúsculas y queda como responsabilidad del programador no modificar su valor a lo largo del programa.

```
In [ ]: CONSTANTE_PI=3.1416
```

En Python nos encontramos con tres grandes grupos de datos: cadenas, números y booleanos. Veremos cada uno de ellos en detalle a continuación.

Cadenas

Las cadenas en Python podemos escribirlas tanto con comillas simples como comillas dobles.

```
In [25]: cadena='Esto es una cadena.'  
cadena2="Esto también es una cadena."  
print cadena, cadena2
```

```
Esto es una cadena Esto también es una cadena
```

Cuando queremos escribir cadenas que ocupan más de una línea usamos tres comillas dobles o simples al comienzo de la cadena:

```
In [26]: poema = """  
          Volverán las oscuras golondrinas  
          en tu balcón sus nidos a colgar,  
          y otra vez con el ala a sus cristales  
          jugando llamarán.  
          """  
print poema
```

```
Volverán las oscuras golondrinas  
en tu balcón sus nidos a colgar,  
y otra vez con el ala a sus cristales  
jugando llamarán.
```

Dentro de las cadenas podemos utilizar caracteres especiales como `\n` que representa el salto de línea o `\t` que inserta un tabulador al mostrar la cadena por pantalla.


Una cadena puede estar precedida por la letra `u` (unicode) o por la letra `r` (del inglés raw, cruda). Las cadenas raw se caracterizan porque los caracteres escapados (`\n` `\t`) no se sustituyen por sus contrapartidas. Esto es especialmente útil para las expresiones regulares.

```
In [43]: raw=r'\t Fin de la línea.\n Comenzamos algo nuevo.'
normal="\t Fin de la línea. \n Comenzamos algo nuevo."
print raw
print normal
```

```
\t Fin de la línea.\n Comenzamos algo nuevo.
      Fin de la línea.
Comenzamos algo nuevo.
```

Por otra parte, las cadenas unicode son cadenas codificadas mediante UTF-8 que nos permiten escribir signos de puntuación o la letra ñ del castellano. Podemos convertir fácilmente la codificación de una cadena binaria en Unicode con la función **Unicode()**.

```
In [1]: #una cadena normal de Python
binstring = ' cadena unicode \xf1'
print binstring
#cadena unicode
unicodestring=u' cadena unicode \xf1'
print unicodestring
```

```
cadena unicode 
cadena unicode ñ
```

```
In [11]: #para transformar una cadena unicode a una codificación tal como utf-8 pa
utfstring=unicodestring.encode('utf-8')
print utfstring
```

```
Æ cadena unicode ñ
```

```
In [13]: #el camino inverso lo realizamos con unicode
unicode(utfstring, 'utf-8')
```

```
Out[13]: u' \u018e cadena unicode \xf1'
```

Mencionar que con el operador + podemos concatenar dos cadenas. ¡OJO! Sólo funciona con strings.

Con las cadenas podemos utilizar una serie de funciones que nos serán muy útiles a la hora de trabajar con ellas, donde s es una cadena o una variable que contiene una cadena:

Función	Descripción	Ejemplo
s.lower()	Convierte todas las letras de una cadena a minúsculas	<code>"BienVenidos".lower()</code> #la salida es bienvenidos
s.upper()	Convierte todas las letras de una cadena a mayúsculas	<code>"minúsculas".upper()</code> # la salida es MINÚSCULAS
s.strip()	Nos devuelve una cadena donde se han eliminado los caracteres blancos iniciales y finales	<code>" cadena limpia ".strip()</code> la salida es 'cadena limpia'
s.isalpha()	Nos indica si es una cadena que contine solo letras	<code>"letras".isalpha()</code> #la salida sería TRUE
s.isdigit()	Nos indica si la cadena se trata de un número convertido a cadena	<code>"1234".isdigit()</code> #la salida sería TRUE
s.startswith()/s.endswith()	Indica si la cadena comienza o termina con un prefijo/subfijo concreto	<code>"submarino".startswith('sub')</code> #la salida sería TRUE
s.find()	Localiza una subcadena dentro de la cadena y devuelve la primera posición	<code>"Juan Garcia".find('ua')</code> # la salida sería 1.
s.replace('original','sustituo')	Reemplaza en la cadena la subcadena 'original' por la cadena 'sustituto'	<code>"cadena sustituye".replace('sustituye', 'nuevo')</code> # la salida es 'cadena nuevo'
s.split('delimitador')	Divide una cadena en tantas cadenas separadas por un delimitador	<code>"c1,c2,c3".split(',')</code> # la salida seria ['c1', 'c2', 'c3']
s.join(lista)	Funciona de forma contraria a split, une una lista de cadenas usando la cadena que lo invoca como delimitador	<code>','.join(['c1', 'c2', 'c3'])</code> #la salida seria "c1,c2,c3"

Como ya hemos visto la función **print** nos permite mostrar por pantalla el valor de una variable. Realmente print recibe como parámetro una cadena. En caso de recibir otro tipo de datos lo transforma automáticamente a cadena.

En ocasiones nos va a interesar imprimir más de una variable en la misma línea. Podemos apoyarnos en la función **format()** para ese caso. Por ejemplo, supongamos que tenemos dos variables `uno` y `dos` y queremos mostrarlas en la misma línea:

```
In [34]: uno=15
dos=20
print "El valor de uno es {0} y el valor de dos es {1}".format(uno,dos)

El valor de uno es 15 y el valor de dos es 20
```

Como vemos en la cadena añadimos {} seguido de un número. Luego añadimos al final de la cadena la llamada a format que incluye en sus parentesis una lista de variables. El número contenido entre los paréntesis indica la posición de la variable que se tiene que mostrar en ese punto de la cadena en la lista de variables de format.

Podríamos usar más de una vez {número} para mostrar en la misma cadena la misma variable más de una vez:

```
In [23]: print "El valor de uno es {0}, el valor de dos es {1}. Como uno vale {0}

El valor de uno es 15, el valor de dos es 20. Como uno vale 15 es cinco
```

Números

Dentro de los **números** disponemos de:

Tipo	Definición	Ejemplo
Entero	Simple	a=12
Long	Añadir L al final	a=123456L
Octal	Añadir 0 al principio	octal=027
Hexadecimal	Añadir 0x al principio	hex=0xFA12
Decimal	Añadiendo un punto	dec=3.1416
Coma flotante	Definición científica	cf=.12e-3
Complejos	Describiendo parte entera y real	complejo= 1.2+5j

```
In [12]: #ejemplo de números
a=-12
b=56
l=123545L
octal=0234
hexadecimal=0xFA12
decimal=3.1416
cientifico=1.2e-13
complejo=1.3+3j
```

Ejercicio: comprueba los tipos de datos de las variables anteriores utilizando la función type en la siguiente celda.

Operadores aritméticos:

Operador	Descripción	Ejemplo
+	Suma	<code>a = 10 + 5 #a es 15</code>
-	Resta	<code>a = 12 - 7 # a es 5</code>
-	Negación	<code>a = -5 #a es -5</code>
*	Multiplicación	<code>a = 7 * 5 # a es 35</code>
**	Exponente	<code>a = 2 ** 3 # a es 8</code>
/	División	<code>a = 12.5 / 2 # a es 6.25</code>
//	División entera	<code>a = 12.5 / 2 # a es 6.0</code>
%	Módulo	<code>a = 27 % 4 # a es 3</code>

Debemos tener en cuenta que para Python cualquier operación entre dos números de un tipo concreto debe devolver el resultado en ese tipo. Por ejemplo, si hacemos una división entre enteros el resultado será un número entero, como en el siguiente ejemplo:

```
In [17]: 3/2
```

```
Out[17]: 1
```

Si queremos que el resultado contenga los decimales deberemos realizar una conversión de tipo con la función **float()** de algunos de los operandos. En caso de existir varios tipos de datos entre los operandos se elige el de mayor representación:

```
In [16]: float(3)/2
```

```
Out[16]: 1.5
```

```
In [2]: #Ejemplo: cálculo del área y la longitud de la circunferencia
radio=12.3
diametro=radio*2
pi=3.1416
longitud=2*radio*pi
area=pi*(radio**2)
print "El circulo con radio {0}, tiene un diámetro de {1}, su longitud de
El circulo con radio 12.3, tiene un diámetro de 24.6, su longitud de ár
```

Ejercicio: El teorema de Pitágoras dice *En todo triángulo rectángulo el cuadrado de la hipotenusa es igual a la suma de los cuadrados de los catetos*, matemáticamente $a^2 + b^2 = c^2$

Siendo $a=15$ y $b=8$, calcula el valor de la hipotenusa (c) en la siguiente celda.

Ejercicio: Supongamos que disponemos de 100€ invertidos en un plazo fijo que rentan cada año un 5%. Es decir, después un año obtenemos $100+5\%$ de $100=100 \times 1.05=105\text{€}$. Tras dos años sería $100 \times 1.05 \times 1.05=110.25$. ¿Cuál será la cantidad de dinero acumulada tras 9 años?. Cálculalo con ayuda de Python en la siguiente celda.

Booleanos

Las variables booleanas sólo pueden tener dos valores: True (cierto) y False (falso). Este tipo de variables y las expresiones que se derivan de ellas son especialmente importante en las estructuras de control. Formalmente se consideran un subtipo de los números. Pero vamos a tratarlos a parte.

Las operaciones fundamentales sobre booleanos son:

Operación	Descripción	Ejemplo
and	¿se cumple a y b?	<code>r=True and False #r es False</code>
or	¿se cumple a ó b?	<code>r=True or False #r es True</code>
not	Lo opuesto de a, No a	<code>r= not True #r es False</code>

Además, como ya hemos comentado, los booleanos son el resultado de muchas expresiones condicionales:

Operación	Descripción	Ejemplo
==	¿son iguales a y b?	<code>5==3 # es False</code>
!=	¿son distintos a y b?	<code>5!=3 # es True</code>
<	¿es a menor que b?	<code>5 < 3 # es False</code>
>	¿es a mayor que b?	<code>5 > 3 # es True</code>
<=	¿es a menor o igual que b?	<code>3<=3 # es True</code>
>=	¿es a mayor o igual que b?	<code>5>=3 #es True</code>

Fechas

Python no tiene un tipo de datos básico para gestionar las fechas. Sin embargo es un tipo de datos muy común cuando analizamos datos. Así que es importante conocer como funciona.

En Python tenemos dos librerías que nos ayudan a gestionar las fechas: `datetime` y `time`.

El objeto `time` dentro de `datetime` se encarga de las horas

```
In [1]: import datetime
#Creamos una fecha usando números
t = datetime.time(1, 2, 3)
print t
print 'hora :', t.hour
print 'minutos:', t.minute
print 'segundos', t.second
print 'microsegundos:', t.microsecond
print 'tzinfo:', t.tzinfo
```

```
01:02:03
hora : 1
minutos: 2
segundos 3
microsegundos: 0
tzinfo: None
```

Por otra parte con `datetime` también podemos gestionar los días. Para crear una fecha concreta usamos `date`. Si además queremos almacenar no sólo la fecha sino también la hora usamos `datetime`. La función `today` devuelve la fecha de hoy. Y podemos mostrarlo con diferentes formatos:

```
In [11]: #creamos una fecha
d1 = datetime.date(2008, 3, 12)
#Creamos una fecha con hora
d3= datetime.datetime(2016,5,14)
print 'd1:', d1
print 'd3:', d3
#cambiamos el año
d2 = d1.replace(year=2009)
print 'd2:', d2
#jugamos con la fecha de hoy
today = datetime.date.today()
print today
print 'ctime:', today.ctime()
print 'tupla:', today.timetuple()
print 'ordinal:', today.toordinal()
print 'Año:', today.year
print 'Mes :', today.month
print 'Día: ', today.day
```

```
d1: 2008-03-12
d3: 2016-05-14 00:00:00
d2: 2009-03-12
2017-09-23
ctime: Sat Sep 23 00:00:00 2017
tupla: time.struct_time(tm_year=2017, tm_mon=9, tm_mday=23, tm_hour=0,
ordinal: 736595
Año: 2017
Mes : 9
Día: 23
```

En ocasiones vamos a querer trabajar con cierta aritmética entre las fechas, por ejemplo calculando desde una fecha determinada los días anteriores o posteriores. Para ello nos podemos apoyar en la función `timedelta`.


```
In [5]: print "microsegundos:", datetime.timedelta(microseconds=1)
print "milisegundos:", datetime.timedelta(milliseconds=1)
print "segundos      :", datetime.timedelta(seconds=1)
print "minutos       :", datetime.timedelta(minutes=1)
print "horas         :", datetime.timedelta(hours=1)
print "días          :", datetime.timedelta(days=1)
print "semanas       :", datetime.timedelta(weeks=1)
```

```
microsegundos: 0:00:00.000001
milisegundos: 0:00:00.001000
segundos      : 0:00:01
minutos       : 0:01:00
horas         : 1:00:00
días          : 1 day, 0:00:00
semanas       : 7 days, 0:00:00
```

```
In [8]: #Aritmética de fechas
hoy= datetime.date.today()
print 'Hoy      :', hoy

un_dia = datetime.timedelta(days=1)
print 'Un día :', un_dia

ayer = hoy - un_dia
print 'Ayer:', ayer

manana = hoy + un_dia
print 'Mañana:', manana

#Comparamos fechas y horas
print 'Horas:'
t1 = datetime.time(12, 55, 0)
print '\tt1:', t1
t2 = datetime.time(13, 5, 0)
print '\tt2:', t2
print '\tt1 < t2:', t1 < t2

print 'Fechas:'
d1 = datetime.date.today()
print '\td1:', d1
d2 = datetime.date.today() + datetime.timedelta(days=1)
print '\td2:', d2
print '\td1 > d2:', d1 > d2
```

```
Hoy      : 2017-09-23
Un día : 1 day, 0:00:00
Ayer: 2017-09-22
Mañana: 2017-09-24
Horas:
      t1: 12:55:00
      t2: 13:05:00
      t1 < t2: True
Fechas:
      d1: 2017-09-23
      d2: 2017-09-24
      d1 > d2: False
```

Por último, nos puede interesar mostrar las fechas con un formato distinto del que nos muestra por defecto (el sistema ISO). Así que podemos usar la función `strftime` o `strptime` para elegir como mostrar una fecha. Además usaremos `strptime` para transformar cadenas de texto en objetos `datetime`.

```
In [9]: format = "%a %b %d %H:%M:%S %Y"

today = datetime.datetime.today()
print 'ISO      :', today

s = today.strftime(format)
print 'strftime:', s

d = datetime.datetime.strptime(s, format)
print 'strptime:', d.strftime(format)

ISO      : 2017-09-23 10:00:28.897000
strftime: Sat Sep 23 10:00:28 2017
strptime: Sat Sep 23 10:00:28 2017
```

```
In [13]: fecha=datetime.datetime.strptime("1-09-2017", "%d-%m-%Y")
print type(fecha)
print fecha

<type 'datetime.datetime'>
2017-09-01 00:00:00
```

Estructuras de datos

Los tipos de datos que hemos visto con anterioridad son tipos de datos simples. En este apartado veremos la colecciones, que son agrupaciones de otros tipos de datos. Dentro de Python tenemos tres estructuras de datos o colecciones básicas:

- Tuplas
- Tuplas
- Diccionarios

Listas

Las listas son una colección ordenada de elementos. En otros lenguajes se conocen como arrays o vectores.

Para definir las listas basta con enumerar sus elementos entre corchetes.

```
In [27]: colores=['rojo','amarillo','azul','verde']
type(colores)
```

Out[27]: list

```
In [19]: #todos los elementos de una lista no tienen por qué ser del mismo tipo
mixta=['Alicia',1,True,'Antonio',36,False,'Lucia',33,'Casper',11]
type(mixta)
```

Out[19]: list

```
In [22]: #podemos anidar listas dentro de listas
anidada=[1,['a','b'],True,[False,'cadena',33]]
type(anidada)
```

Out[22]: list

Con la función len podemos saber el número de elementos contenidos en una lista.

```
In [29]: print "La lista Mixta tiene {0} elementos. La lista colores tiene {1} ele
La lista Mixta tiene 10 elementos. La lista colores tiene 4 elementos
```

Para acceder a los elementos de un vector usamos el nombre del vector seguido de los corchetes y el índice del elemento. Las listas comienzan en 0 en python.

```
In [25]: print colores[2]
print anidada[1][1]
print anidada[3]

azul
b
[False, 'cadena', 33]
```

Python permite la notación "slicing", que nos permite acceder a un subconjunto de elementos de una lista definiendo el subrango de índices mediante [inicio:fin]. Por ejemplo, en la lista colores:

rojo	amarillo	azul	verde
0	1	2	3
-4	-3	-2	-1

Tenemos que:

```
In [30]: #los dos elementos centrales
print colores[1:3]
#desde segundo elemento
print colores[1:]
#todos los elementos del vector
print colores[:]
# desde azul elemento en la posición 3, hasta el final ya que el último i
print colores[2:100]

['amarillo', 'azul']
['amarillo', 'azul', 'verde']
['rojo', 'amarillo', 'azul', 'verde']
['azul', 'verde']
```

Alternativamente podemos usar los índices de forma negativo, comenzando a contar desde la última posición como -1 hasta la primera posición de la lista. Siguiendo con el ejemplo de la lista colores, tenemos que:

```
In [32]: #Se refiere a el penúltimo elemento
print colores[-2]
#Selecciona todos los elementos desde la posición -3 hasta el inicio; es
print colores[:-3]
#Selecciona todos los elementos desde la posición -3 hasta el final de la
print colores[-3:]

azul
['rojo']
['amarillo', 'azul', 'verde']
```

Podemos añadir un nuevo elemento al final una lista usando el operador **+** o borrar un elemento de una lista usando el operador **del()**.

```
In [34]: #añadimos rosa a la lista de colores
colores=colores+"rosa"
print colores

['rojo', 'amarillo', 'azul', 'verde', 'rosa']
```

```
In [35]: #borramos azul de la lista de colores
del colores[2]
print colores

['rojo', 'amarillo', 'verde', 'rosa']
```

Además disponemos de las siguientes funciones sobre listas:

Función	Descripción	Ejemplo
<code>l.append(elemento)</code>	Añade un sólo elemento al final de la lista.	<code>[1,2,3].append(4)</code> #salida <code>[1,2,3,4]</code>
<code>l.insert(elemento, index)</code>	Añade un elemento en la posición que se indica.	<code>[2,4,6].insert(3,1)</code> #salida <code>[2,3,4,6]</code>
<code>l.index(elemento)</code>	Busca un elemento en la lista y devuelve su índice.	<code>[2,4,6].index(6)</code> #salida <code>2</code>
<code>l.remove(elemento)</code>	Borra un elemento de la lista.	<code>[2,4,6].remove(4)</code> #salida <code>[2,6]</code>
<code>l.sort()</code>	Ordena la lista en su lugar	<code>[4,2,8,6].sort()</code> #salida <code>[2,4,6,8]</code>
<code>l.reverse()</code>	Invierte la lista en lugar	<code>[2,4,6].reverse()</code> #salida <code>[6,4,2]</code>
<code>l.pop(indice)</code>	Borra y devuelve el elemento en la posición indicada	<code>[2,4,6].pop(1)</code> #salida <code>4</code> , la lista quedaría <code>[2,6]</code>

Un detalle interesante es que el tipo `string` es a su vez considerado una lista especial donde todos sus elementos son caracteres. Así que las funciones que hemos visto para listas así como los mecanismos de acceso mediante índices positivos y negativos también se pueden usar con listas.

Tuplas

Las tuplas son un tipo especial de listas, a las que se les aplica los mismos mecanismos, con la diferencia de que se definen entre paréntesis. Realmente el constructor de las tuplas es la coma, que concatena los elementos, pero por claridad se suele indicar que se usen los paréntesis.

Su principal diferencia con respecto a las listas es que son inmutables. Una vez creadas no se pueden modificar.

```
In [20]: dias_semana=('lunes','martes','miércoles','jueves','viernes','sabado','do
print type(dias_semana)
print "Hoy es {}".format(dias_semana[1])
no_es_tupla=(1)
print type(no_es_tupla)
es_tupla=(1,)
print type(es_tupla)
print "Semana laboral: {}".format(dias_semana[0:4])

<type 'tuple'>
Hoy es martes
<type 'int'>
<type 'tuple'>
Semana laboral: ('lunes', 'martes', 'mi\x03\xa9rcoles', 'jueves')
```

Ejercicio: En el ejemplo anterior al imprimir miércoles aparece sin acento, ¿cómo solucionaríamos este problema?. Define de nuevo días de la semana evitando ese problema de codificación e imprímelo.

Diccionarios

Los diccionarios son colecciones formadas por pares valor-clave o también conocidas como matrices asociativas. Se define como una enumeración entre llaves de pares separados por dos puntos, donde el primer valor es la clave y el segundo el valor asociado. Las claves deben ser tipos de datos simples o tuplas (datos inmutables), pero los valores pueden ser cualquier tipo de datos: datos simples, otro diccionario, listas, tuplas, objetos,....

Podremos acceder a los valores de los elementos de la colección usando como índice la clave. Y podremos añadirlos directamente escribiendo la clave y el valor.

```
In [2]: diccionario= {"España": 'Madrid', 'Francia': u"París", 'Reino Unido': 'Lo
print type(diccionario)

<type 'dict'>
```

```
In [3]: #leemos un elemento
print diccionario['Francia']
#añadimos un nuevo elemento
diccionario['Alemania']='Berlín'
print diccionario
#modificamos un elemento
diccionario['Austria']='Viena'
print diccionario

París
{'Francia': u'Par\xeds', 'Reino Unido': 'Londres', 'Espa\x03\xbla': 'Ma
{'Francia': u'Par\xeds', 'Reino Unido': 'Londres', 'Espa\x03\xbla': 'Ma
```

Los principales métodos que podemos usar con los diccionarios son:

- **d.keys()** devuelve la lista de claves de un diccionario.
- **d.items()** devuelve una lista con pares clave-valor como tuplas.
- **d.values()** devuelve una lista con los valores del diccionario.

```
In [5]: print diccionario.keys()
print diccionario.items()
print diccionario.values()

['Francia', 'Reino Unido', 'Espa\xc3\xbla', 'Austria', 'Italia', 'Alema
[('Francia', u'Par\xeds'), ('Reino Unido', 'Londres'), ('Espa\xc3\xbla',
[u'Par\xeds', 'Londres', 'Madrid', 'Viena', 'Roma', 'Berl\xc3\xaddn']
```

Estructuras de Control

Sentencias condicionales

La sentencia condicional más básica que encontramos en Python es `if` seguido de la condición a evaluar, que debe ser una expresión booleana según hemos visto en el curso, seguida de dos puntos. El bloque de sentencias que se ejecutará en caso de que la condición sea cierta debe estar indentado.

```
In [17]: comida='sopa'
if comida=='sopa':
    print "Muchas gracias, pero hoy no me apetece."
    print "¿Hay hamburguesas?"
```

Muchas gracias pero hoy no me apetecen.

```
In [18]: #Cuidado con la indentación de los bloques
comida='sopa'
if comida=='sopa':
    print "Muchas gracias, pero hoy no me apetece."
print "¿Hay hamburguesas?"
```

Muchas gracias pero hoy no me apetecen.
¿Hay hamburguesas?

Si deseamos añadir una alternativa a la condición principal usamos la palabra reservada `else` seguida de dos puntos.

```
In [14]: #EJECUTA ESTE CÓDIGO VARIAS VECES PARA VER SU RESULTADO
#Llamamos a una biblioteca: random
import random
semaforo=['VERDE','ROJO','AMBAR','VERDE','ROJO','AMBAR']
#El método shuffle modifica el orden de los elementos del array
random.shuffle(semaforo)
print semaforo
if semaforo[0]=='VERDE':
    print "Vamos a cruzar"
else:
    print "Esperemos hasta que se ponga verde"
```

['AMBAR', 'VERDE', 'VERDE', 'ROJO', 'AMBAR', 'ROJO']
Esperemos hasta que se ponga verde

En Python carecemos de la instrucción `switch` como en otros lenguajes así que cuando existen varias condiciones asociadas al `if` usamos la palabra reservada `elif`.

```
In [16]: #EJECUTA ESTE CÓDIGO VARIAS VECES PARA VER SU RESULTADO
#Llamamos a una biblioteca: random
import random
semaforo=['VERDE','ROJO','AMBAR','VERDE','ROJO','AMBAR']
#El método shuffle modifica el orden de los elementos del array
random.shuffle(semaforo)
print semaforo
if semaforo[0]=='VERDE':
    print "Vamos a cruzar"
elif semaforo[0]=='AMBAR':
    print "No sé si deberíamos cruzar, igual con prudencia"
elif semaforo[0]=='ROJO':
    print "Esperemos hasta que se ponga verde"
else:
    print "Vaya, el semáforo está raro"

['AMBAR', 'ROJO', 'VERDE', 'AMBAR', 'VERDE', 'ROJO']
No sé si deberíamos cruzar, igual con prudencia
```

Por último tenemos un constructor de sentencias condicionales similar a ? en otros lenguajes, lo que nos permite escribir sentencias condicionales en una sola línea. Se suelen construir con la sintaxis `A if C else B`, que sería equivalente a `if C then A, else B`.

```
In [18]: numero=1024
es_par= 'Si' if numero%2==0 else 'No'
print es_par

Si
```

Bucles

Dentro de Python nos encontramos con dos estructuras de control iterativas: `while` `for ... in`.

Los bucles `while` se ejecutan mientras se cumpla la condición indicada.

```
In [22]: dia=0;
while dia<7:
    if dia<5:
        print "Hoy es {0}. ¡A trabajar!".format(dias_semana[dia])
    else:
        print "Hoy es {0}. Es fin de semana, buen descanso".format(dias_s
dia=dia+1

Hoy es lunes. ¡A trabajar!
Hoy es martes. ¡A trabajar!
Hoy es miércoles. ¡A trabajar!
Hoy es jueves. ¡A trabajar!
Hoy es viernes. ¡A trabajar!
Hoy es sabado. Es fin de semana, buen descanso
Hoy es domingo. Es fin de semana, buen descanso
```

Es importante escribir una condición que se cumpla en algún momento o entraremos en un bucle infinito. Sin embargo, en ocasiones nos interesa crear bucles infinitos. En ese caso usaremos la instrucción `break` para romper el bucle.

```
In [23]: dia=0
while True:
    if dia<5:
        print "Hoy es {0}. ¡A trabajar!".format(dias_semana[dia])
    elif dia<7:
        print "Hoy es {0}. Es fin de semana, buen descanso".format(dias_s
    else:
        break;
    dia=dia+1
```

```
Hoy es lunes. ¡A trabajar!
Hoy es martes. ¡A trabajar!
Hoy es miércoles. ¡A trabajar!
Hoy es jueves. ¡A trabajar!
Hoy es viernes. ¡A trabajar!
Hoy es sabado. Es fin de semana, buen descanso
Hoy es domingo. Es fin de semana, buen descanso
```

Los bucles `for ... in ...` se utilizan para iterar de forma genérica sobre una colección de elementos. Es decir, se usan para recorrer listas, diccionarios,...

Entre la palabra reservada `for` e `in` usamos una variable que nos va a permitir acceder en bloque del bucle al elemento actual en la iteración. Tras la palabra reservada `in` incluimos la colección que deseamos recorrer.

```
In [24]: for dia in dias_semana:
        print dia
```

```
lunes
martes
miércoles
jueves
viernes
sabado
domingo
```

```
In [32]: for item in diccionario.values():
        print item
```

```
París
Londres
Madrid
Viena
Roma
Berlín
```

En ocasiones no queremos recorrer una lista, sino queremos un índice; es decir una variable numérica cuyo valor vamos incrementando. En ese caso usamos la función `xrange(inicio, fin, incremento)`. Esta función nos genera automáticamente una lista con la secuencia de números que necesitamos.

Existe una función equivalente, llamada `range`. Sin embargo, en la versión 2.7 por eficiencia es preferible la función `xrange`.


```
In [34]: for i in xrange(1,10,2):  
         print i
```

```
1  
3  
5  
7  
9
```

Funciones

Una función es un bloque de instrucciones con un objetivo claro, que realizar una serie de operaciones y siempre devuelve un valor. En Python no disponemos de métodos, siempre se devuelve un valor. Si en una función no se indica que valor se debe retornar, por defecto devuelve el objeto None.

En Python las funciones se definen con la palabra reservada `def` seguida del nombre de la función y los parámetros entre paréntesis.

```
In [36]: def suma(a, b):  
         """Esta función se encarga de sumar a con b y devolver el valor """  
         return a+b  
  
print suma(2,3)
```

```
5
```

Si escribimos un comentario justo después de la primera línea de nuestra función entre *tres comillas dobles*, esa cadena se considerará el docstring de la función. Es el texto que se nos mostrará cuando busquemos ayuda relacionada con la función. Es por eso que es importante documentar apropiadamente las funciones que definamos.

Dada una función podemos mostrar su ayuda mediante la función **help(funcion)** o bien con el nombre de la función seguida de `?`

```
In [38]: help(suma)
```

```
Help on function suma in module __main__:
```

```
suma(a, b)  
    Esta función se encarga de sumar a con b y devolver el valor
```

Cuando llamamos a las funciones podemos usar los argumentos en el mismo orden en el que se han definido en la función o podemos alterar el orden indicando que valor queremos asociar a cada parámetro.

```
In [39]: def resta(a,b):  
         """Esta función realiza la resta de a menos b"""  
         return a-b  
  
print resta(5,3)  
print resta(b=5,a=3)
```

```
2  
-2
```

También es posible, no obstante, definir funciones con un número variable de argumentos, o bien asignar valores por defecto a los parámetros para el caso de que no se indique ningún valor para ese parámetro al llamar a la función.

```
In [44]: def potencia(exponente, base=2):  
        """ Esta función eleva a la potencia a la base. Si no se indica la ba  
        return base**exponente  
  
        print potencia(10)  
        print potencia(3, 5)  
  
1024  
125
```

Si quisieramos crear una función que recibiera un número variable de parámetros usaríamos la * delante del último parámetro. Este último parámetro funcionaría como una tupla.

```
In [45]: def tsuma(a, *otros):  
        """Esta función realiza la suma de todos los parámetros que recibe"""  
        suma=a  
        for item in otros:  
            suma=suma+item  
        return suma  
  
        print tsuma(2, 3)  
        print tsuma(2, 3, 4, 5, 6)  
  
5  
20
```

En Python se permite la devolución de más de un objeto mediante la instrucción return. Podemos recoger los diversos objetos devueltos por una función mediante la asignación múltiple.

```
In [3]: def circulo(radio):  
        """Función que calcula el área y la longitud de un círculo"""  
        diametro=radio*2  
        area=2*radio*pi  
        longitud=pi*(radio**2)  
        return area, longitud, diametro  
  
d, a, l=circulo(13)  
print "Para un círculo con radio 13, tenemos un diametro de {0}, su área  
Para un círculo con radio 13, tenemos un diametro de 81.6816, su área e:
```

Si queremos usar un diccionario en lugar de una tupla con este último parámetro solo hay que añadir `**` en lugar de sólo un `*`. En este caso las claves serán el nombre de las variables pasadas como parámetro.

En las funciones existen dos formas de paso de parámetros:

- *Por valor*, donde los cambios que sufre la variable dentro del método no son visibles una vez finalizada la ejecución de la función.
- *Por referencia*, donde en lugar de una copia de la variable se pasa una referencia a la zona de memoria donde se almacena el dato. Por tanto los cambios en la variable son visibles más allá de la función.

Los parámetros en Python se pasan todos por valor excepto los objetos que se pasan por referencias. Esto quiere decir que, a excepción de los objetos inmutables, todos los cambios en los objetos serán visibles más allá del método.

```
In [4]: def visible(variable,objeto):
        """Función ejemplo para detectar la visibilidad de una variable simple
        variable=variable+3
        objeto.append(23)
        print variable,objeto

x=2
y=[22]
visible(x,y)
#la variable x al pasar por valor no conservará los cambios
#pero y como es una lista si conservará los cambios
print x,y

5 [22, 23]
2 [22, 23]
```

En Python todo es un objeto, así que podemos asignar funciones a una variable y ejecutarlas cuando nos sean útiles.

```
In [53]: def funcion(t):
        """Función ejemplo para pasarla como función"""
        return "Función simple,{0} ".format(t)

def ejecutar(f,*params):
    """Ejecuta una función que pasamos por parámetro, junto con sus argumentos"""
    return f(params)

fvar=funcion
print type(fvar)
print ejecutar(fvar,"prueba")

<type 'function'>
Función simple,('prueba',)
```

Ficheros

Podemos abrir los ficheros directamente con el método **open(nombre, modificador)**.

- Nombre es la ruta al fichero.
- Modificador es el modo de apertura del fichero:
 - r: lectura
 - w: escritura
 - a: añadir

Podemos escribir en un fichero con el método **write()**.

Podemos leer un fichero completo con el método **readlines()** y si deseamos almacenarlo en un string con el método **read()** .

Al terminar de usarlo debemos cerrarlo con **close()**.

```
In [4]: #abrimos el fichero
f=open('ElAmigoManso.txt','r')
texto=f.read()
print texto[0:500]
f.close()
```

The Project Gutenberg EBook of El amigo Manso, by Benito Pérez Galdós

This eBook is for the use of anyone anywhere at no cost and with almost no restrictions whatsoever. You may copy it, give it away or re-use it under the terms of the Project Gutenberg License included with this eBook or online at www.gutenberg.org/license

Title: El amigo Manso

Author: Benito Pérez Galdós

Release Date: September 16, 2017 [EBook #55563]

Language: Spanish

Character set encoding: UTF-8

*** START OF

```
In [8]: f=open("nuevofichero.txt",'w')
f.write(texto[0:500])
f.close
#vemos que hemos escrito
fp=open("nuevofichero.txt",'r')
print fp.read()
fp.close()
```

The Project Gutenberg EBook of El amigo Manso, by Benito Pérez Galdós

This eBook is for the us

Ficheros JSON

JSON es un formato popular para el intercambio de información entre sistemas informáticos a través de servicios web, entre otras tareas.

- Para trabajar con ficheros JSON tenemos que importar la librería `json`.
- La función **`dumps(data)`** formatea una cadena con el formato json.
- La función **`loads(json)`** transforma nuestro json en tipos de datos base de Python (cadenas, listas, tuplas,...).

Una vez tenemos los datos extraídos del formato JSON podemos acceder a los datos mediante sus índices como si fuera una lista algo más compleja.

```
In [10]: import json
datos={"agenda":[{"nombre":"John Doe","edad":30,"telefono": "672891345"}],

json_codificado=json.dumps(datos)

json_decodificado=json.loads(json_codificado)

print json_decodificado["agenda"][1]["nombre"]
```

Mary Merry

Excepciones

Es posible que al trabajar con programas tengamos que lidiar con excepciones.

Para capturar las excepciones debemos encerrar el bloque de sentencias que podría generar la excepción entre las palabras reservadas `try` y `except`. Añadimos tantas sentencias `except` como excepciones queremos capturar seguidas del nombre de la excepción y un bloque de sentencias que se encarga de manejar la excepción. Siempre es buena idea dejar una última sentencia `except` sin especificar la excepción para capturar cualquier excepción adicional que pueda surgir. Si se produce una excepción se ejecutará primero el bloque de instrucciones que acompaña el `except` asociado a la excepción o bien si no hay un bloque específico el global.

En caso de que no haya excepción, no se ejecuta ningún bloque `except`. Se ejecutaría en ese caso el bloque de sentencias que acompañan la etiqueta `else`.

Por último el bloque `finally` son un conjunto de instrucciones que se ejecuta sí o sí haya o no haya excepción.

```
In [17]: try:
        num = int("3a")
        print no_existe
except NameError:
    print "La variable no existe"
except ValueError:
    print "El valor no es un numero"
```

El valor no es un numero

```
In [16]: import sys
try:
    f=open('fichero-no-existe.txt')
    s = f.readline()
    i=int(s.strip())
except OSError as err:
    print("Error del sistema operativo: {0}".format(err))
except IOError as err:
    print("Error: no se ha encontrado el fichero -> {0}".format(err))
except ValueError:
    print("No se ha podido formatear el número de forma adecuada")
except:
    print("Error inesperado ",sys.exc_info()[0])
    raise
else:
    f.close()
finally:
    print "Esto se ejecuta si o si pase lo que pase"
```

Error: no se ha encontrado el fichero -> [Errno 2] No such file or dire
Esto se ejecuta si o si pase lo que pase

Algunas excepciones bastante comunes:

Excepción	Descripción
BaseException	Clase de la que heredan todas las excepciones.
ArithmeticError	Error aritmético
AttributeError	No se ha encontrado un atributo en el objeto
FloatingPointError	Error en coma flotante
OverflowError	Valor demasiado grande para manejarlo en el programa
IOError	Error en una operación de entrada/salida
OSError	Error en una llamada al sistema operativo
IndexError	Index fuera de rango para una colección
KeyError	La clave del diccionario no existe

