Aprendiendo Python

Contenido

Introducción

Python es un lenguaje de programación de propósito general muy fácil de aprender, con una sintaxis característica que hace que los programas escritos en él sean muy legibles, ampliamente utilizado por grandes empresas y programadores independientes; y además, libre.

Python fue creado a finales de la década de los 80 por Guido van Rossum, un programador holandés.

¿Por qué Python?

- Fácil de aprender
- Altamente expresivo
- Sintáxis legible
- Software libre
- Baterías incluidas
- Gran cantidad de bibliotecas para diversos propósitos

Python es un lenguaje que está diseñado para que sea elegante y sencillo de escribir, y a la vez provee herramientas de muy alto nivel para la manipulación de estructuras de datos. Existe, además, una gama muy amplia de bibliotecas gratuitas para prácticamente cualquier propósito. Todo esto hace de Python un lenguaje muy versátil y práctico.

Características del lenguaje

- **Multiparadigma:** Además de ser un lenguaje imperativo como muchos otros, Python provee la capacidad para programar según distintos paradigmas, como por ejemplo la *Programación Orientada a Objetos* (POO) o incluso el paradigma funcional.
- **Sistema de tipos:**

    - Tipado dinámico (dynamic typing): El tipo de las variables y las expresiones es evaluado en tiempo de ejecución. No es necesario declarar las variables antes de usarlas.
    - Tipado fuerte (strong typing): El intérprete no permite operaciones entre tipos de datos distintos sin convertirlos explícitamente.
- **Facilidad de extensión:** Se pueden escribir nuevos módulos fácilmente en C o C++. Y a su vez, es posible invocar al intérprete de Python desde C o C++.

El Zen de Python

Los usuarios de Python se refieren a menudo a la Filosofía Python que es bastante análoga a la filosofía de Unix. El código que sigue los principios de Python de legibilidad y transparencia se dice que es "pitónico" (*pythonic* en inglés)". Contrariamente, el código opaco u ofuscado es bautizado como "no pitónico" (*unpythonic*). Estos principios fueron famosamente descritos por el desarrollador de Python Tim Peters en *El Zen de Python*.

```
Bello es mejor que feo.
Explícito es mejor que implícito.
Simple es mejor que complejo.
Complejo es mejor que complicado.
Plano es mejor que anidado.
Disperso es mejor que denso.
La legibilidad cuenta.
Los casos especiales no son tan especiales como para quebrantar las reglas.
Aunque lo práctico gana a la pureza.
Los errores nunca deberían pasar silenciosamente.
A menos que hayan sido silenciados explícitamente.
Frente a la ambigüedad, rechaza la tentación de adivinar.
Debería haber una (y preferiblemente sólo una) manera obvia de hacerlo.
Aunque esa manera puede no ser obvia al principio, a menos que seas holandés.
Ahora es mejor que nunca.
Aunque nunca es a menudo mejor que *ya mismo*.
Si la implementación es difícil de explicar, es una mala idea.
Si la implementación es fácil de explicar, puede que sea una buena idea.
Los espacios de nombres (namespaces) son una gran idea ¡Hagamos más de eso!
```

El intérprete de Python

Python incluye un intérprete interactivo en el cual se escriben las instrucciones en una especie de línea de comandos; las expresiones pueden ser introducidas una a una, pudiendo verse el resultado de su evaluación inmediatamente, lo que da la posibilidad de probar porciones de código en el modo interactivo antes de integrarlo como parte de un programa. Esto resulta útil tanto para las personas que se están familiarizando con el lenguaje como para los programadores más avanzados.

Para abrir la consola interactiva de Python, basta con ejecutar:

```
$ python
```

La cónsola interactiva de python puede evaluar expresiones del lenguaje python, incluyendo operaciones aritméticas. Es también posible asignar valores a variables, y los valores permanecerán en memoria mientras el intérprete esté ejecutándose.

```
Python 2.5.2 (r252:60911, Oct  5 2008, 19:29:17)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 1 + 1
```

```
2
>>> a = range(10)
>>> print a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> type(a)
<type 'list'>
```

También es posible importar otros módulos desde el intérprete interactivo:

```
>>> import math
```

Para explorar los atributos (incluyendo métodos) utilizamos la función `dir()`

```
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan',
'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh', 'trunc']
>>> math.pi
3.141592653589793
```

Tipos básicos

Python implementa los tipos de datos habituales en otros lenguajes, como los tipos numéricos `int` y `float`, así como el tipo lógico o `bool`. Para los valores nulos, se utiliza el valor `None`.

Es posible convertir de un tipo a otro invocando explícitamente el tipo deseado, siempre que la conversión sea válida.

```
>>> str(1)
'1'
>>> int('2')
2
>>> bool(1)
True
```

Merecen especial atención el tipo *string* y los tipos estructurados o "colecciones", dentro de los cuales existen *diccionarios*, *tuplas*, *listas* y conjuntos.

Cadenas de texto (`str`)

Los strings son cadenas de texto que pueden definirse de varias formas:

- Entre comillas simples: `'hola mundo!'`
- Entre comillas dobles: `"hola mundo!"`
- Entre comillas triples (cadenas multi-línea):

```
'''hola
mundo'''

"""todo
bien?"""
```

Concatenación

Es posible concatenar dos o más cadenas usando el operador +, o usando el método `''.join()`. Para determinar la longitud de una cadena, se utiliza la función `len()`.

```
>>> a = "hola"
>>> a += " mundo!"
>>> a
'hola mundo!'
>>> len(a)
10
>>> b = ''.join(["x", "y", "z", a])
>>> b
'xyzhola mundo'
>>> '_'.join(["cadena", "compuesta", "con", "delimitador"])
'cadena_compuesta_con_delimitador'
```

Formateo

Es posible "formatear" cadenas usando el operador %:

```
>>> "La respuesta es %s." % 42
'La respuesta es 42.'
>>> "El monto (bs %f) no es suficiente" % 64.85
'El monto (bs 64.85) no es suficiente"
>>> "El precio del producto seleccionado es de bs %.2f" % 50.4625
'El precio del producto seleccionado es de bs 50.46'
```

También es posible formatear cadenas con el método `format()` del tipo `str`, con la siguiente sintaxis:

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # a partir de python 2.7
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')   # los indices pueden repetirse
'abracadabra'
```

Repetición

Una cadena puede repetirse utilizando el mismo operador de multiplicación (*)

```
>>> h = "hola"
>>> h * 3
'holaholahola'
```

Indexación

Para acceder a cualquiera de los caracteres de la cadena, se indexa de la misma manera que un "arreglo" en la mayoría de los lenguajes. El primer caracter corresponde al índice `0`.

```
>>> "Venezuela"[5]
'u'
```

También pueden utilizarse índices negativos, los cuales comienzan a recorrer la cadena desde el último caracter.

```
>>> "Venezuela"[-1]
'a'
>>> "Venezuela"[-2]
'l'
```

"Slicing"

Es posible obtener una sub-cadena de un string especificando un rango en el índice, a esto se le conoce como "rebanado" o *slicing*.

```
>>> a = "Venezuela"
>>> a[2:4]
'ne'
>>> a[:4]
'Vene'
>>> a[4:]
'zuela'
>>> a[:]
'Venezuela'
```

Pertenencia (`in`)

Para determinar si un caracter o subcadena está contenido dentro de otra cadena, se utiliza el operador *in*:

```
>>> "zuel" in "Venezuela"
True
>>> "b" in "Venezuela"
False
```

Booleanos

Los datos de tipo *booleano* pueden tomar los valores `True` o `False` (así, con la primera letra en mayúscula). Y las expresiones lógicas pueden evaluarse utilizando los operadores *and*, *or*, y *not* y los operadores usuales de comparación (<, >, <=, >=, ==, !=).

```
>>> a = "hola mundo"
>>> a[:4] == "hola" and a[5:] == "mundo"
True
```

Colecciones

Listas

Las listas son uno de los tipos de datos fundamentales en Python. A diferencia de los arreglos en otros lenguajes, las listas de Python son dinámicas y permiten una serie de operaciones bastante útiles.

Para definir una lista, se colocan los elementos entre corchetes. Una lista puede contener cualquier tipo de datos, incluyendo otras listas.

```
>>> a = [1, 2, 3, "hola", ['x', 'y']]
```

range()

Una función del lenguaje bastante útil es `range(n)`, la cual genera una lista de enteros dentro del intervalo `[0, n)`:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

También puede invocarse especificando ambos límites:

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
```

E incluso el tamaño del incremento:

```
>>> range(10, 20, 3)
[10, 13, 16, 19]
```

Operaciones sobre listas

Todas las operaciones previamente definidas para el tipo `str` aplican para las listas:

```
>>> a = [1, 2, 3, "hola", ['x', 'y']]
>>> len(a)
5
>>> ['x', 'y'] in a
True
>>> a[1]
2
>>> a[2:4]
[3, 'hola']
>>> a * 2
[1, 2, 3, 'hola', ['x', 'y'], 1, 2, 3, 'hola', ['x', 'y']]
>>> [1, 2, 3] + [4, 5] + ["string"]
[1, 2, 3, 4, 5, 'string']
```

Adicionalmente, se definen los siguientes métodos específicos para las listas:

append()

```
>>> li = ["a", "b", "c"]
>>> li.append("d")
>>> li
['a', 'b', 'c', 'd']
```

extend()

```
>>> st = ["e", "f"]
>>> li.extend(st)
>>> li
['a', 'b', 'c', 'd', 'e', 'f']
```

index()

```
>>> li = ["a", "b", "c"]
>>> li.index('c')
2
>>> li.index("f")
Traceback (most recent call last)_
        File "<stdin>", line 1, in <module>
ValueError: 'f' is not in list
```

remove()

```
>>> li.remove('c')
>>> li
['a', 'b', 'd', 'e', 'f']
```

Listas por comprensión

Una de las características más poderosas de Python es la posibilidad de definir *listas por comprensión*. Ésta es una característica muy propia de lenguajes funcionales.

La ventaja de las listas por comprensión radica en la posibilidad de definir una colección de elementos de una manera acorde a definiciones matemáticas. Por ejemplo, para generar una lista con todos los enteros impares hasta 99:

```
>>> L = [x for x in range(100) if x % 2 != 0]
>>> L
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31,
33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61,
63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91,
93, 95, 97, 99]
```

Las listas por comprensión pueden contener expresiones complejas y funciones anidadas:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

Tuplas

Una tupla es una estructura parecida a una lista, con la diferencia de que ésta es *inmutable*, es decir, no pueden eliminarse o agregarse elementos, ni éstos pueden cambiar una vez creada la tupla.

Las tuplas se definen como una secuencia de elementos separados por comas, y encerrados entre paréntesis.

```
>>> t = ("tuples", "are", "immutable")
>>> t[0]
'tuples'
>>> t[0] = "assignment"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Las operaciones de pertenencia, indexación y "slicing" funcionan de igual forma que en las listas:

```
>>> t = (10, 11, 12)
>>> 10 in t
True
>>> t[-1]
12
>>> t[1:]
(11, 12)
```

Una funcionalidad bastante útil en el caso de las tuplas, es la posibilidad de hacer asignaciones múltiples a un conjunto de variables.

```
>>> x, y, z = (7, 8, 9)
>>> x
7
>>> y
8
>>> z
9
```

Las tuplas se usan en los casos en los que se sabe que los datos no necesitarán modificarse. Sin embargo, siempre es posible convertir de uno a otro tipo de dato con las funciones nativas `list()` y `tuple()`.

```
>>> t = ("x", "y", "hola")
>>> list(t)
['x', 'y', 'hola']
>>> tuple(range(10))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Diccionarios

Un diccionario define una relación 1 a 1 entre claves y valores. Se define como un conjunto de pares "<clave>: <valor>" entre llaves.

```
>>> d = {"servidor": "posma", "database": "master"}
>>> d
{'servidor': 'posma', 'database': 'master'}
>>> d["servidor"]
'posma'
>>> d["database"]
'master'
>>> d["posma"]
>>> d = {"servidor": "posma", "database": "master"}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'posma'
```

Los valores de los diccionarios pueden ser de cualquier tipo, incluso otros diccionarios. De igual forma, un diccionario puede contener simultáneamente valores de distintos tipos. Las claves deben ser de algún tipo *inmutable*, como números, cadenas o incluso tuplas.

Es importante acotar que los diccionarios en Python no siguen ningún tipo de orden entre sus elementos.

```
>>> d = {1: "uno", 2: "dos", 2.5: "dos punto cinco"}
>>> d[2.5]
'dos punto cinco'
```

Para eliminar un registro en el diccionario se utiliza la función `del(k)`:

```
>>> del(d[2.5])
>>> d
{1: "uno", 2: "dos"}
```

Para limpiar el contenido completo de un diccionario, se utiliza el método `clear()`.

```
>>> d.clear()
>>> d
{}
```

Un par de funciones nativas de considerable utilidad son `zip()` y `dict()`. La función `zip(L1, L2)` retorna una lista de tuplas correspondiendo los valores de L1 y L2 respectivamente, y la función dict(L) recibe una lista de tuplas y retorna un diccionario que corresponde a dichas tuplas. Por ejemplo:

```
>>> ciudades = ["Caracas", "Berlin", "Buenos Aires", "Lima"]
>>> paises = ["Venezuela", "Alemania", "Argentina", "Peru"]
>>> parejas = zip(paises, ciudades)
>>> parejas
[('Venezuela', 'Caracas'), ('Alemania', 'Berlin'), ('Argentina', 'Buenos Aires'),
('Peru', 'Lima')]
>>> capitales = dict(parejas)
>>> capitales
{'Argentina': 'Buenos Aires', 'Venezuela': 'Caracas', 'Peru': 'Lima', 'Alemania': 'Berlin'}
```

Conjuntos

Python tiene la particularidad de implementar `set` como tipo de dato nativo, el cual corresponde al concepto matemático de conjunto, e implementa todas sus funciones básicas.

Un conjunto se define como una secuencia de elementos entre llaves, o mediante la función `set(L)` a partir de una lista de elementos L.

```
>>> pares = {2, 4, 6, 8, 10}
>>> pares
set([8, 10, 4, 2, 6])
type(pares)
<type 'set'>
>>> impares = set([n for n in range(10) if n % 2 != 0])
>>> impares
set([1, 3, 9, 5, 7])
```

Correspondiendo con el principio matemático de los conjuntos, ningún elemento se repite.

```
>>>set([1, 2, 3, 2, 1])
set([1, 2, 3])
```

Un conjunto puede definirse a partir de una cadena de texto, y es útil para saber cuáles caracteres existen en un string.

```
>>> zen = "If the implementation is hard to explain, it's a bad idea."
>>> set(zen)
set(['a', ' ', 'b', 'e', 'd', "'", 'f', 'I', 'h', 'm', 'l', 'p', 'n', 'i', 's',
'r', 't', 'x', '.', ',', 'o'])
```

El tipo `set` implementa los siguientes métodos:

add()

```
>>> conj = {1, 2, 3}
>>> conj.add(4)
>>> conj
set([1, 2, 3, 4])
```

clear()

```
>>> conj.clear()
>>> conj
set([])
```

copy()

```
>>> conj.add(1)
>>> conj.add(42)
>>> conj2 = conj.copy()
>>> conj2
set([1, 42])
```

Copiar un conjunto no es lo mismo que asignárselo a otra variable, ya que en la asignación ambas variables se refieren a un mismo objeto, por lo que modificaciones a uno afectarían el valor del otro y viceversa.

difference()

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

En lugar de usar el método `difference()`, podemos utilizar el operador `-`:

```
>>> x - y
set(['a', 'e', 'd'])
>>> x - y - z
set(['a', 'e'])
```

discard()

Si un elemento se encuentra en un conjunto, este método lo elimina, y si el elemento no existe, no sucede nada.

```
>>> x = {"a","b","c","d","e"}
>>> x.discard("a")
>>> x
set(['c', 'b', 'e', 'd'])
>>> x.discard("z")
>>> x
set(['c', 'b', 'e', 'd'])
```

remove()

A diferencia de `discard()`, `remove()` elimina un elemento dado, y si éste no existe ocurre un `KeyError`.

```
>>> x = {"a","b","c","d","e"}
>>> x.remove("a")
>>> x
set(['c', 'b', 'e', 'd'])
>>> x.remove("z")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'z'
```

intersection()

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

union()

```
>>> c1 = {"Carlos", "Jorge", "Luis"}
>>> c2 = {"Oscar", "Antonio"}
>>> c1.union(c2)
set(['Luis', 'Antonio', 'Jorge', 'Carlos', 'Oscar'])
```

isdisjoint()

Retorna `True` si la intersección entre dos conjuntos es nula.

```
>>> set1 = {"a", "b", "c"}
>>> set2 = {"c", "d", "e"}
>>> set3 = {"d", "e", "f"}
>>> set1.isdisjoint(set2)
False
>>> set1.isdisjoint(set3)
True
```

issubset() / issuperset()

```
>>> x = {"a", "b", "c", "d", "e"}
>>> y = {"c", "d"}
>>> x.issubset(y)
False
>>> y.issubset(x)
True
>>> x.issuperset(y)
True
```

También es posible utilizar los operadores de comparación para evaluar subconjuntos:

```
>>> set1 = {1, 2, 3}
>>> set2 = {2}
>>> set3 = set2.copy()
>>> set2 < set1
True
set2 == set3
True
set1 <= set2
False
```

pop()

El método `pop()` retorna un elemento (el primero que encuentra) y lo elimina del conjunto. Se produce un `KeyError` cuando el conjunto se encuentra vacío.

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

Usando un archivo fuente

El intérprete interactivo es bastante útil para hacer pruebas y entender el lenguaje, pero para el desarrollo de software es necesario ejecutar *scripts*. Para esto necesitamos escribir nuestro código en un archivo fuente, el cual llamaremos *script.py*. La extensión `.py` no es necesaria, pero es parte de una convención.

Para editar un archivo de texto podemos utilizar cualquier aplicación. En Linux, por ejemplo, podemos utilizar *vim*.

```
$ vim script.py
```

Esto abre la interfaz del editor de texto. Para comenzar a insertar texto, presionamos la tecla "i". Para más información sobre los comandos de vim, revisar la documentación oficial.

Comenzaremos escribiendo un programa de prueba en el editor:

```
# -*- coding: utf-8 -*-
import math

""" De esta manera definimos los
```

```
comentarios multilínea"""
print "el valor de pi es %s" % math.pi
```

El caracter numeral (#) para insertar comentarios de una línea. En el caso de utilizar caracteres especiales dentro del archivo (como la palabra "multilínea" que está acentuada), se coloca un comentario en la primera línea que le indica al intérprete la codificación a utilizar (utf-8).

Para salir del modo de inserción, guardar el archivo y salir del editor, presionamos secuencialmente: *Esc*, *:*, `wq` y *Enter*.

Para ejecutar el script, llamamos al intérprete pasándole el nombre del archivo a ejecutar.

```
$ python script.py
el valor de pi es 3.14159265359
```

Funciones

Como la mayoría de los lenguajes, Python provee el uso de funciones, las cuales se declaran mediante la palabra reservada `def`, seguido del nombre de la función, una lista opcional de parámetros entre paréntesis y dos puntos (:):

```
def mi_funcion(param1, param2, param3=False):
        pass    # mi_funcion no hace nada
```

- Para una función, es posible definir parámetros opcionales, especificándoles un valor por defecto.
- No se especifica ningún tipo de valor de retorno, ya que en Python los tipos se determinan dinámicamente.

Indentación

En Python la indentación de bloques de código es obligatoria, ya que no se utilizan las llaves (`{}`) ni ningún otro delimitador para determinar el comienzo y fin de los bloques. Esta es una de las características que garantizan la legibilidad del código. La convención recomienda que en lugar del caracter de tabulación, se configure el editor para que inserte 4 espacios por indentación.

Como podemos ver, tampoco se terminan las instrucciones con punto y coma ni ningún otro caracter especial.

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

Docstring

Para documentar una función, colocamos un comentario multilínea al comienzo del cuerpo de una función. Abriremos nuevamente el archivo `script.py` y definimos la función `factorial` con su respectiva documentación.

```
# -*- coding: utf-8 -*-

def factorial(n):
    """
    Definición recursiva de factorial
```

```
    ================================

    Retorna el factorial de un entero n
    si n == 1 retorna 1
    sino retorna n * fact(n-1)
    """
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```

Si guardamos los cambios y abrimos el intérprete interactivo (llamada a `python` sin argumentos), podemos importar nuestro módulo (script.py):

```
>>> import script
```

Ahora con la función `help()` podemos revisar la documentación de la función factorial.

```
>>> help(script.factorial)
```

Si queremos poder invocar `factorial()` sin el prefijo del módulo, debemos importar la función explícitamente.

```
>>> from script import factorial
>>> factorial(7)
5040
```

Estructuras de control de flujo

Python cuenta con las estructuras de control de flujo usuales: if, while, for.

if

El condicional `if` se utiliza como en la mayoría de los lenguajes imperativos. También se utiliza `elif` para encadenar condicionales.

```
def calcular_impuestos(ingreso):

    if ingreso <= 8004:
        impuesto = 0
    elif ingreso <= 13469:
        y = (ingreso - 8004.0)/10000.0
        impuesto = (912.17 * y + 1400) * y
    elif ingreso <= 52881:
        z = (ingreso - 13469.0) / 10000.0
        impuesto = (228.74 * z + 2397.0) * z + 1038.0
    else:
        impuesto = ingreso * 0.44 - 15694

        return impuesto
```

while

El siguiente código lee un caracter por teclado y se sale del ciclo (usando `break`) cuando el caracter leído es un salto de línea.

```
import sys

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

print "Input: %s" % text
```

Esta lectura de caracteres, sin embargo, puede efectuarse utilizando la función nativa `raw_input()`.

Una particularidad de los ciclos en python, es que pueden incluir un bloque `else`, el cual se ejecuta si el programa sale limpiamente (sin usar break).

```
while condicion:
if error():
    # manejar error
    break    # salir del ciclo
else:
    # no hubo error
    hacer_algo()
```

for

El ciclo `for` en python, a diferencia de lenguajes como C o Java, es más bien una iteración entre los elementos de una secuencia (conocido en otros lenguajes como *foreach*). También acepta opcionalmente un bloque `else` que se ejecuta cuando no ocurrió un `break` dentro del ciclo:

```
def contiene_par(lista):
    for n in lista:
        if n % 2 == 0:
            # se encontró un número par
            return True
    else:
        # no se encontró un número par
        return False
```

Para implementar un `for` con un contador entero como es usual en los lenguajes imperativos, se utiliza `range(n)`.

```
for n in range(10):
    print n
```

Es posible recorrer cualquier colección en un ciclo `for`, incluyendo diccionarios, de la siguiente manera:

```
for key,val in d:
    print "d[%s] => %s" % (key, val)
```

Excepciones y manejo de archivos

Excepciones

Las excepciones son un elemento fundamental del lenguaje Python, y su uso es fuertemente aconsejado en *El Zen de Python*: "Los errores nunca deberían pasar silenciosamente."

Otra de las filosofías de Python es que "es mejor pedir disculpas que pedir permiso". Es decir, es recomendable asumir que un bloque de código puede generar excepciones y atajarlas, en lugar de hacer verificaciones antes de ejecutar el bloque.

Algunos ejemplos de excepciones comunes son:

- Acceder a una clave inexistente en un diccionario genera un `KeyError`.
- Buscar el índice de un elemento inexistente en una lista genera un `ValueError`
- Invocar un método inexistente genera un `AttributeError`.
- Hacer referencia a una variable inexistente genera un `NameError`.
- Tratar de operar sobre tipos de datos mezclados sin conversión explícita genera un `TypeError`.

Para prever y manejar excepciones, se coloca el código dentro de un bloque `try-except` de la siguiente manera:

```
while True:
    try:
        n = raw_input("Introduzca un entero: ")
        n = int(n)
        break
    except ValueError:
        print("El valor introducido es inválido, por favor intente de nuevo")
```

Es posible manejar por separado diversos tipos de excepciones, encadenando los bloques `except` de la siguiente manera:

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print "I/O error({0}): {1}".format(e.errno, e.strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

El manejo de excepciones también permite una cláusula `else`, para definir código que debe ejecutarse en caso de no ocurrir ninguna excepción. Esto funciona de la misma forma que `finally` en Java.

```
try:
    alguna_operacion()
except Exception as e:
    print "Error al intentar la operación: %s" % e
else:
```

```
    print "Operación exitosa"
```

Archivos

Para esta sección, podemos intentar abrir el mismo archivo `script.py` sobre el cual estábamos trabajando, e intentemos imprimir cada una de sus líneas en el intérprete interactivo:

```
>>> fobj = open("script.py", "r")
>>> for line in fobj:
...     print line.rstrip()
```

El segundo argumento `"r"` indica que el archivo se está abriendo en modo lectura. Una vez terminamos de utilizar un archivo, es necesario cerrarlo:

```
>>> fobj.close()
```

Para escribir sobre un archivo utilizamos el método `write()`. El siguiente código toma el archivo `script.py` y copia cada una de las líneas escribiéndolas sobre otro archivo precedido del número de línea:

```
try:
    fobj_in = open("script.py", "r")
    fobj_out = open("lineas.txt","w")
    i = 1
    for line in fobj_in:
        print line.rstrip()
        fobj_out.write(str(i) + ": " + line)
        i = i + 1
    fobj_in.close()
    fobj_out.close()
except IOError as err:
    print "Error en manejo de archivo: %s" % er
```

Ejercicio práctico

Implementar un script que lea un archivo en donde en cada línea habrá una cadena de bits, y escriba en un archivo de salida su correspondiente entero decimal en cada línea.