# Taller de Python

Posma Group

8 de febrero de 2014

# Contenido


1. Aprendiendo Python...
2. Características del lenguaje
3. Tipos Básicos
4. Colecciones
5. Usando un archivo fuente
6. Estructuras de control de flujo
7. Excepciones y manejo de archivos

# ¿Por qué Python?

- Fácil de aprender

# ¿Por qué Python?

- Fácil de aprender
- Altamente expresivo

# ¿Por qué Python?

- Fácil de aprender
- Altamente expresivo
- Sintáxis legible

# ¿Por qué Python?

- Fácil de aprender
- Altamente expresivo
- Sintáxis legible
- Software libre

# ¿Por qué Python?

- Fácil de aprender
- Altamente expresivo
- Sintáxis legible
- Software libre
- Baterías incluidas

# ¿Por qué Python?

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

# Características

- Multiparadigma

# Características

- Multiparadigma
  - Programación imperativa
  - Programación funcional
  - Programación Orientada a Objetos

# Características

- Multiparadigma
  - Programación imperativa
  - Programación funcional
  - Programación Orientada a Objetos
- Facilidad de extensión

# Características

- Multiparadigma
  - Programación imperativa
  - Programación funcional
  - Programación Orientada a Objetos
- Facilidad de extensión
- Sistema de tipos:
  - Tipado dinámico
  - Tipado fuerte

# Sistema de tipos

## Tipado dinámico

El tipo de las variables y las expresiones es evaluado en tiempo de ejecución. No es necesario declarar las variables antes de usarlas.

# Sistema de tipos

## Tipado dinámico

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

El intérprete no permite operaciones entre tipos de datos distintos sin convertirlos explícitamente.

# El intérprete interactivo

- 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.

# El intérprete interactivo

- 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.

## Para iniciar el intérprete...

```
$ python
```

# El intérprete interactivo

```
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...
>>> 1 + 1
2
```

# El intérprete interactivo

```
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...
>>> 1 + 1
2

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

# El intérprete interactivo

```
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...
>>> 1 + 1
2

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

>>> type(a)
<type 'list'>
```

# Filosofía Python

## 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).

Estos principios están descritos en *El Zen de Python*, al cual podemos echar un vistazo ejecutando la siguiente línea en el intérprete:

```
>>> import this
```

# El intérprete interactivo

También es posible importar otros módulos desde el intérprete interactivo:

```
>>> import math
```

# El intérprete interactivo

También es posible importar otros módulos desde el intérprete interactivo:

```
>>> import math
```

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

```
>>> dir(math)
...
```

# El intérprete interactivo

También es posible importar otros módulos desde el intérprete interactivo:

```
>>> import math
```

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

```
>>> dir(math)
...

>>> 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`.

# 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.

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

# Cadenas de Texto

## Definición

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?"""
```

# Cadenas de Texto

## Concatenación de cadenas

Es posible concatenar dos o más cadenas usando el operador +, o usando el método `''.join()`

```
>>> a = "hola"
>>> a += " mundo!"
>>> a
'hola mundo!'
```

# Cadenas de Texto

## Concatenación de cadenas

Es posible concatenar dos o más cadenas usando el operador +, o usando el método ''.join()

```
>>> a = "hola"
>>> a += " mundo!"
>>> a
'hola mundo!'

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

# Cadenas de Texto

## Longitud de una cadena

La longitud de una cadena puede obtenerse mediante la función nativa `len()`:

```
>>> len("Caracas")
7
```

# Cadenas de Texto

## Longitud de una cadena

La longitud de una cadena puede obtenerse mediante la función nativa `len()`:

```
>>> len("Caracas")
7
```

## Formateo de cadenas

También es posible "formatear" cadenas usando el operador %:

```
>>> "La respuesta es %s." % 42
'La respuesta es 42.
>>> "El valor de %s es %s" % (obj.name, obj.val)
>>> "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'
```

# Cadenas de Texto

## Repetición

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

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

# Cadenas de Texto

## 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

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

# Cadenas de Texto

## Indexación negativa

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

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

# Cadenas de Texto

## *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'
```

# Cadenas de Texto

## Pertenencia

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

## Definición

- Los datos de tipo `bool` pueden tomar los valores `True` o `False`.
- 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
```

# Booleanos

## Definición

- Los datos de tipo `bool` pueden tomar los valores `True` o `False`.
- 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
```

Además, son considerados como "falsos" los siguientes valores:

- `None`
- El número cero en cualquier tipo
- Cadenas o colecciones vacías: `''`, `()`, `[]`, $\{\}$

# Listas

## Definición

- La lista es uno de los tipos de dato fundamentales en Python.
- Una lista puede contener cualquier tipo de datos, incluyendo otras listas.

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

# Listas

## Definición

- La lista es uno de los tipos de dato fundamentales en Python.
- Una lista puede contener cualquier tipo de datos, incluyendo otras listas.

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

Podemos convertir una cadena en una lista con la función `list()`:

```
>>> list('hola')
['h', 'o', 'l', 'a']
```

# Listas

## range()

`range(n)` es una función nativa que genera una lista de enteros dentro del intervalo $[0, n)$:

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

# Listas

## range()

`range(n)` es una función nativa que 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]
```

# Listas

## range()

`range(n)` es una función nativa que 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]
```

# Listas

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

```
>>> a = [1, 2, 3, "hola", ['x', 'y']]
>>> len(a)
5
```

# 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
```

# 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']
```

# 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']
```

# Métodos de listas

Además de las operaciones básicas, las listas implementan varios métodos propios.

# Métodos de listas

Además de las operaciones básicas, las listas implementan varios métodos propios.

## append()

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

# Métodos de listas

Además de las operaciones básicas, las listas implementan varios métodos propios.

## 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']
```

# Métodos de listas

## 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
```

# Métodos de listas

## 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 = ['a', 'b', 'c']
>>> li.remove('b')
>>> li
['a', 'c']
```

# Listas por comprensión

- Una de las características más poderosas de Python es la posibilidad de definir listas por comprensión.
- Es posible definir una colección de elementos de una manera acorde a una definición matemática.

# Listas por comprensión

- Una de las características más poderosas de Python es la posibilidad de definir listas por comprensión.
- Es posible definir una colección de elementos de una manera acorde a una definición matemática.

Ejemplo: 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]
```

# Listas por comprensión

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']
```

# Listas por comprensión

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']
```

Lista de números primos:

```
>>> noprimes = [j for i in range(2, 8) for j in range(i*2, 50, i)]
>>> primes = [x for x in range(2, 50) if x not in noprimes]
>>> print primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

# Tuplas

## Definición

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.

# Tuplas

## Definición

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.

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

# Tuplas

## Operaciones

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)
```

# Tuplas

## Asignación múltiple

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

# Tuplas

## Conversión entre listas y tuplas

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

## Definición

Un diccionario define una relación 1 a 1 entre claves y valores, algo muy parecido a los objetos de la clase Hashtable en Java.

```
>>> d = {"servidor": "posma", "database": "master"}
>>> d
{'servidor': 'posma', 'database': 'master'}
>>> d["servidor"]
'posma'
>>> d["database"]
'master'
>>> d["posma"]
```

# Diccionarios

- Los valores de los diccionarios pueden ser de cualquier tipo, incluso otros diccionarios.

# Diccionarios

- Los valores de los diccionarios pueden ser de cualquier tipo, incluso otros diccionarios.
- Un diccionario puede contener simultáneamente valores de distintos tipos.

# Diccionarios

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

# Diccionarios

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

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

# Diccionarios

## del()

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

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

# Diccionarios

## del()

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

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

## clear()

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

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

# Diccionarios

## zip() y dict()

```
>>> 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')]
```

# Diccionarios

## zip() y dict()

```
>>> 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

## Definición

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.

# Conjuntos

## Definición

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])
```

# Conjuntos

## Definición

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'>
```

# Conjuntos

## Definición

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])
```

# Conjuntos

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

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

# Conjuntos

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.

```
>>> 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'])
```

# Métodos de Conjuntos

## add()

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

# Métodos de Conjuntos

## add()

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

## clear()

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

# Métodos de Conjuntos

## copy()

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

# Métodos de Conjuntos

## 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.

# Métodos de Conjuntos

## 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'])
```

# Métodos de Conjuntos

## 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'])


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

# Métodos de Conjuntos

## discard()

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

# Métodos de Conjuntos

## 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'
```

# Métodos de Conjuntos

## intersection()

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

# Métodos de Conjuntos

## 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'])
```

# Métodos de Conjuntos

## 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
```

# Métodos de Conjuntos

## issubset()

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

# Métodos de Conjuntos

## issubset()

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

## issuperset()

```
>>> x.issuperset(y)
True
```

# Métodos de Conjuntos

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

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

# Métodos de Conjuntos

## 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'
```

# Creando 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.

# Creando 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.

Crearemos un archivo usando `vim`:

```
$ vim script.py
```

# Creando un archivo fuente

Escribamos un código de prueba:

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

""" De esta manera definimos los
comentarios multilinea"""
print "el valor de pi es %s" % math.pi
```

# Creando un archivo fuente

Escribamos un código de prueba:

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

""" De esta manera definimos los
comentarios multilinea"""
print "el valor de pi es %s" % math.pi
```

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

```
$ python script.py
```

# Creando un archivo fuente

Escribamos un código de prueba:

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

""" De esta manera definimos los
comentarios multilinea"""
print "el valor de pi es %s" % math.pi
```

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

## Definición

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
```

# Funciones

## Definición

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
```

- Es posible definir parámetros opcionales, especificándoles un valor por defecto.

# Funciones

## Definición

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
```

- 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.

# Funciones

## 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.

# Funciones

## 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.

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

# Funciones

## 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.

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

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

# Funciones

## Docstring

Para documentar una función, colocamos un comentario multilínea al comienzo del cuerpo de una función.

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

def factorial(n):
    """
    Definicion 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)
```

# Funciones

Desde el intérprete interactivo, podemos importar ahora nuestro módulo (script.py):

```
>>> import script
```

# Funciones

Desde el intérprete interactivo, podemos importar ahora nuestro módulo (script.py):

```
>>> import script
```

## help()

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

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

# Funciones

Desde el intérprete interactivo, podemos importar ahora nuestro módulo (script.py):

```
>>> import script
```

## help()

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
```

# Condicional

## if-else

```
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
```

# Ciclos

## while

```
import sys

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

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

# Ciclos

## while

```
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 puede hacerse utilizando la función `raw_input()`

# Ciclos

## while-else

Una particularidad de los ciclos en python, es que pueden incluir un bloque `else`, el cual se ejecuta si el ciclo termina sin usar `break`.

```
while condition:
    if error_occurred():
        # manejar error
        break    # salir del ciclo
    handle_true()
else:
    # la condicion ya es falsa, se ejecuta el siguiente bloque
    handle_false()
```

# Ciclos

## 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. El ciclo `for` también acepta opcionalmente un bloque `else`:

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

# Ciclos

## for

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
```

# Ciclos

## for

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
```

## Recorriendo un diccionario

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

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

# Práctica

## Ejercicio práctico 1

Los factores primos de 13195 son 5, 7, 13 and 29.

Implementar una función que reciba un entero n y retorne su factor primo más grande.

Éxito!

# Excepciones

*"Los errores nunca deberían pasar silenciosamente."* (*El Zen de Python*)

# Excepciones

*"Los errores nunca deberían pasar silenciosamente."* (*El Zen de Python*)

Algunas excepciones comunes son:

- Acceder a una clave inexistente en un diccionario genera un `KeyError`.

# Excepciones

*"Los errores nunca deberían pasar silenciosamente."* (*El Zen de Python*)

Algunas 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`

# Excepciones

*"Los errores nunca deberían pasar silenciosamente."* (*El Zen de Python*)

Algunas 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`.

# Excepciones

*"Los errores nunca deberían pasar silenciosamente."* (*El Zen de Python*)

Algunas 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`.

# Excepciones

*"Los errores nunca deberían pasar silenciosamente."* (*El Zen de Python*)

Algunas 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`.

# Excepciones

## Capturando excepciones

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

# Excepciones

Es posible manejar por separado varios tipos de excepciones:

```
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
```

# Excepciones

## try-except-else

El bloque de manejo de excepciones acepta opcionalmente un bloque else, el cual se ejecuta si el código no genera ninguna excepción.

```
try:
    alguna_operacion()
except:
    print "Error al intentar la operacion"
else:
    print "Operacion exitosa"
```

# Manejo de archivos

## Definición

Python implementa el manejo de archivos a través del tipo `file`. Para abrir un archivo se utiliza la función nativa `open()`:

# Manejo de archivos

## Definición

Python implementa el manejo de archivos a través del tipo `file`. Para abrir un archivo se utiliza la función nativa `open()`:

```
file = open("archivo.txt", "r")
```

El primer argumento de `open()` es el nombre del archivo que se desea abrir, y el segundo es el modo de acceso, que puede ser:

- "r": lectura
- "w": escritura
- "a": agregación

# Manejo de archivos

## Definición

Python implementa el manejo de archivos a través del tipo `file`. Para abrir un archivo se utiliza la función nativa `open()`:

```
file = open("archivo.txt", "r")
```

El primer argumento de `open()` es el nombre del archivo que se desea abrir, y el segundo es el modo de acceso, que puede ser:

- "r": lectura
- "w": escritura
- "a": agregación

La función también acepta el modo `r+` (lectura y escritura). Si no se especifica un modo, se asume que el archivo está en modo de lectura.

# Manejo de archivos

## Lectura de archivos

Para leer de un archivo, los objetos de tipo `file` cuentan con los siguientes métodos:

- `read()`: Retorna una sola cadena con todo el archivo.

# Manejo de archivos

## Lectura de archivos

Para leer de un archivo, los objetos de tipo `file` cuentan con los siguientes métodos:

- `read()`: Retorna una sola cadena con todo el archivo.
- `readline()`: Va retornando línea por línea.

# Manejo de archivos

## Lectura de archivos

Para leer de un archivo, los objetos de tipo `file` cuentan con los siguientes métodos:

- `read()`: Retorna una sola cadena con todo el archivo.
- `readline()`: Va retornando línea por línea.
- `readlines()`: Retorna una lista de líneas.

# Manejo de archivos

## Lectura de archivos

Para leer de un archivo, los objetos de tipo `file` cuentan con los siguientes métodos:

- `read()`: Retorna una sola cadena con todo el archivo.
- `readline()`: Va retornando línea por línea.
- `readlines()`: Retorna una lista de líneas.

También es posible iterar sobre un archivo, lo cual es rápido, eficiente y simple:

```
f = open()
for line in f:
    print line
```

# Manejo de archivos

Desde el intérprete interactivo, intentemos abrir el mismo archivo script.py.

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

# Manejo de archivos

Desde el intérprete interactivo, intentemos abrir el mismo archivo script.py.

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

Una vez terminamos de utilizar un archivo, es necesario cerrarlo:

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

# Manejo de archivos

## Escritura en un archivo

Para escribir sobre un archivo utilizamos el método `write()`:

```
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" % err
```

# Manejo de archivos

## Escritura en un archivo

Para escribir sobre un archivo utilizamos el método `write()`:

```
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" % err
```

También es posible escribir una lista de líneas de una vez utilizando la función `writelines()`

# Práctica

## Ejercicio práctico 2

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.

Éxito!

*Muchas gracias!*