Programando con Python

POO

POO introducción

- ¿Qué aprenderemos? (a la manera Python) :
  - Encapsulación
  - Abstracción de datos
  - Polimorfismo
  - Herencia

POO introducción

- **Clase**: El esquema o plano de construcción de donde se originan los objetos.
- En Python, todo es un objeto
- Se usa la función type() para determinar que clase origina un objeto
- Clase mínina:

```
class Robot:
    pass
```

POO introducción

- **Atributos**: habilidad específica o características que algo o alguien posee.
- Suele llamársele como propiedades
- En Python propiedades y atributos son algo diferentes
- Aunque los atributos se definen internamente en las clases. en Python, los atributos se puede definir dinámicamente
- Internamente en la clase se pueden verificar con __dict__

POO introducción

- **Métodos**: en Python son esencialmente funciones.
- Definen el comportamiento de los objetos
- Aunque pueden ser definidos fuera de la clase, no es una buena práctica
- El primer parámetro es usado como referencia de la instancia llamada. Es nombrado *self*
- *self* ¡no es una palabra reservada de Python!, usuarios de C++ o Java lo pueden llamar *this*, pero puede ser contraproducente.

POO introducción

- **El método __init__**: es llamado inmediata y automáticamente después que una instancia ha sido creada.
- Ejm:

```
Class A:
    def __init(self)__:
        print("¡¡__init__ ha sido ejecutado!!")
```

POO introducción

```
def __init__(self, radius):
        self.radius = radius
```

- El método **__init__** es conocido como el constructor
- Es invocado cada vez que un objeto es creado en memoria.
- Aparte del parámetro self, se pueden agregar tantos parámetros como se necesite para inicializar un objeto.
- Definir el constructor no es requerido y si no se hace, Python proporciona uno vacío: __init__()

POO introducción

- Definir una clase (ejemplo):

```
import math

class Circle:

    def __init__(self, radius):
        self.radius = radius

    def get_area(self):
        return math.pi * self.radius ** 2

    def get_perimeter(self):
        return 2 * math.pi * self.radius
```

- En Python el parámetro **self** es requerido en cada método y refiere al objeto que invocó tal método
- Al llamar el método no se necesita pasar ningún valor a self.
- Dentro de la clase se usa self, para tener acceso a los atributos y métodos propios del objeto

POO introducción

- **Conceptos:**
- **Encapsulación**: Ocultar o proteger los datos, de manera que solo puedan se accedidos usando funciones especiales, es decir los métodos
- **Ocultamiento de información**: principio de OOP por medio del cual los datos no puede ser cambiada accidentalmente.
- **Abstracción de datos:** Los datos están ocultos y se usó la encapsulación, es decir:

  Abstracción de datos = Encapsulación + Ocultamiento

POO introducción

- **Abstracción de datos:**
- **Métodos getter**: permiten obtener o tener acceso a los valores de los atributos, no cambian el valor del atributo.
- **Métodos setter**: son usados para cambiar los valores de los atributos.

POO introducción

- **Ejemplo:**

```
class Robot:
    def __init(self, name= None):
        self.name = name
    def decir_hola(self):
        if self.name:
            print("Hola, yo soy " + self.name)
        else:
            print("Hola, soy un robot sin nombre")
    def get_name(self):
        return self.name
    def set_name(self, name):
        self.name = name
```

POO introducción

- **El Zen de Python:**
  - se invoca usando import this
  - indica que: “Debe haber una, y preferiblemente solo una, forma obvia de hacer algo”
  - Ya lo trataremos más adelante

POO introducción

- **Los métodos __str__ y __repr__**
- Si son definidos en la clase son usados para obtener una representación personalizada del objeto.
- Si solo se define __str__, __repr__ retornará el valor predefinido
- Si solo se define __repr__, funcionará para llamadas de __repr__ y de __str__
- Considere siempre usar __str__, ya que __repr__ puede que no funcione para objetos diferentes de string, se debe cumplir:

```
obj == eval(repr(obj))
```

- __repr__ se usa para representación interna, __str__ para el usuario final

POO introducción

- **Ejemplo:**

```
class Robot:
    def __init(self, name, build_year):
        self.name = name
        self.build_year = build_year
    def __rpre__(self):
        return "Robot(\"" + self.name + "\", "
               + str(self.build_year) + ")"
    def __str__(self):
        return "Name: " + self.name + ", Build Year: "
               + str(self.build_year)
```

POO introducción

- **Atributos Public, Protected y Private**
- In POO significan:
  - Private, sólo puede ser usado por el propietario
  - Protected, úselo a su propio riesgo, solo por alguien que escriba una subclase
  - Public, puede ser usado sin restricción
- En Python, se pueden definir así:
  - name , Public
  - _name, Protected,
  - __name, Private

  (ver el ejemplo reescrito)

POO introducción

- **Atributos de clase y de instancia**
- In POO significan:
  - de instancia: son particulares a cada instancia
  - de clase: pertenecen a la clase
- En Python, atributo de clase:

```
Class A:
    a = "atributo de clase"

x = A()
y = A()
x.a
```

- OJO: Si se desean cambiar se debe hacer con el NombreClase.atributo, si se hace con un objeto, se crea un atributo de instancia.
(ver ejemplo)

POO introducción

- **Métodos estáticos**
- In POO significan:
  - se acceden desde la propia clase, no de una instancia
- En Python:

```
Class Robot:
    __counter = 0
    def __init(self):
        type(self).__counter += 1

    @staticmethod       # <- ojo con el decorador
    def RobotInstances():
        return Robot.__counter
```

- Simplemente no llevan *self*

POO introducción

- **Métodos de clase**
- Son como métodos estáticos, pero están atados a la clase, la que se pasa como referencia
- En Python:

```
Class Robot:
    __counter = 0
    def __init(self):
        type(self).__counter += 1

    @classmethod
    def RobotInstances(cls):
        return cls, Robot.__counter
```

POO introducción

- **Métodos de clase, cuando usarlos:**
- En la definición de métodos llamados factory, no se tratará el tema de patrones de diseño
- En conjunto con métodos estáticos, que deben llamar a otros métodos estáticos.

  (ver ejemplo)
- Se volverán a tratar mas adelante en herencia.

POO introducción

- **Uso de propiedades:**
- Cuando se desee usar la forma de acceso objeto.atributo, pero respetando el principio de encapsulamiento.
- Se eliminan los prefijos get_ y set_ y se sustituyen por @property (para el get_) y @atributo.setter por el setter, cada método simplemente lleva el nombre del atributo.

POO introducción

- **Uso de propiedades (ejemplo)**

```
class P:
    def __init__(self, x):
        self.x = x

    @property
    def x(self):
        return self.__x

    @x.setter
    def x(self, x):
        if x < 0:
            self.__x = 0
        elif x > 1000:
            self.__x = 1000
        else:
            self.__x = x
```

Fin parte 1

python™

Programando con Python

Clases y Objetos (Herencia y Polimorfismo)

POO introducción

- **Herencia:**
- Python soporta herencia y herencia múltiple
- Sintáxis: class SubclassName(SuperclassName):
  pass
- Funciones type() y isinstance(), la primera nos dice el tipo, la segunda si el objeto es o no derivado de una clase en particular

  (ver ejemplos)

Clases y Objetos: Herencia y Polimorfismo

- Herencia:
  - En Python se usa la función **super()**, para llamar algún método de la super clase
  - Ejemplo:

```
class Rectangle(Shape):

    def __init__(self, length, width):
        super().__init__()
        self.__length = length
        self.__width = width
```

Ver ejemplo inheritance.py / herencia_simple.py

Clases y Objetos: Herencia y Polimorfismo

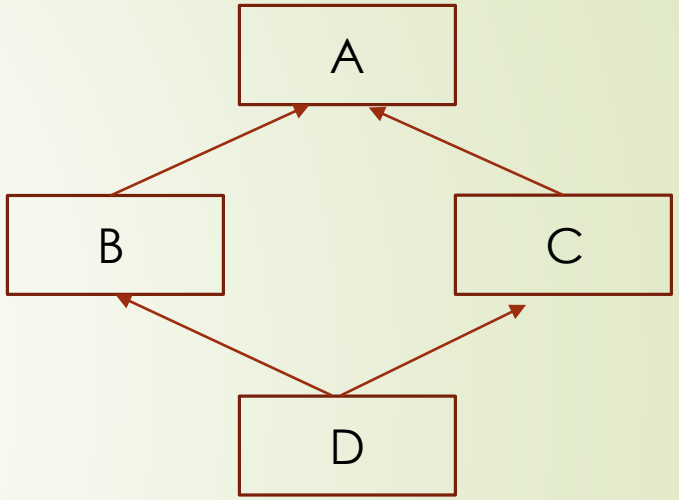
- Herencia múltiple:
  - En Python es posible derivar una clase desde otras clases, esto se llama herencia múltiple.
  - Sintaxis:

```
class ParentClass_1:
    # body de ParentClass_1

class ParentClass_2:
    # body de ParentClass_2

class ParentClass_3:
    # body de ParentClass_1

class ChildClass(ParentClass_1, ParentClass_2, ParentClass_3):
    # body de ChildClass
```

Ver ejemplo: herencia_multiple.py / herencia_multiple2.py

POO introducción

- **Herencia múltiple:**
- Sintaxis: class SubclassName(Superclass1, SuperClass2, …, SuperClassN):
  pass
- Para evitar el problema del "mortal diamante de la muerte", Python usa el llamado MRO (method resolution order), el cual le permite resolver el problema. Ver ejemplo.



POO introducción

- **Herencia – sobreescritura (overriding):**
- Cuando un método de la superclase se implementa de diferente forma en la subclase
- Es posible llamar a métodos de la superclase
- La subclase puede sus propios métodos (especialización)

  (ver ejemplos)

POO introducción

- **Diferencia entre overwriting, overloading y overriding:**
- Overwriting, cuando una función se declara nuevamente con diferentes parámetros
- Overloading, no existe en Python, pero se puede simular con parámetros predefinidos. O mejor aún, usando la forma def(*x):, lo cual indica que la función f puede recibir un número indeterminado de parámetros.
- Overriding, cuando un método de la subclase tiene el mismo nombre de otro en la superclase.

Clases y Objetos: Herencia y Polimorfismo

- Polimorfismo:
  - En Python el polimorfismo es definido de manera que un método de la clase hija tenga el mismo nombre que el de la clase padre.
  - Es en esencia una sobre-escritura
  - Ver ejemplos:
    - method_overriding.py
    - method_overriding_2.py

Clases y Objetos: Herencia y Polimorfismo

- object – la clase base:
  - En Python todas las clases heredan de la clase **object** de forma implícita.
  - Eso quiere decir que esta dos expresiones son equivalentes:

```
class MyClass:
    pass

class MyClass(object):
    pass
```

Clases y Objetos: Herencia y Polimorfismo

- object – la clase base:
  - La clase object posee métodos que son heredados por todas las clases. Algunos importantes son:
    1. __new__() -> crea el objeto
    2. __init__() -> después __new__(), se llama para inicializar los atributos del objeto
    3. __str__() -> retorn a una representación en string del objeto (ver ejemplo __str__method.py), por lo general se sobreescribe según la necesidad

Clases y Objetos: Herencia y Polimorfismo

- Sobreescribiendo la funcionalidad de los operadores:
  - Python permite redefinir la forma como los operadores incluidos definen sus operaciones
  - De allí que el operador **+** sirva tanto para sumar números como para concatenar dos o más strings
  - Los métodos especiales que definen los operadores comienzan y terminan con doble guion bajo, así el de **+** es __add__()
  - Las clases int y la clase str, lo definen cada uno de acuerdo a lo que necesiten hacer y de allí que se pueda usar el mismo signo **+** de dos maneras diferentes.
  - Aunque empiezan con doble guion bajo no son privados, porque también terminan con doble guion bajo

Clases y Objetos: Herencia y Polimorfismo

- Operador y su método especial:

Operador	Método Especial	Descripción
+	__add__(self, object)	Suma
-	__sub__(self, object)	Resta
*	__mul__(self, object)	Multiplicación
**	__pow__(self, object)	Exponenciación
/	__truediv__(self, object)	División
//	__floordiv__(self, object)	División Entera
%	__mod__(self, object)	Modulo
==	__eq__(self, object)	Igual a

Clases y Objetos: Herencia y Polimorfismo

- Operador y su método especial:

Operador	Método Especial	Descripción
!=	__ne__(self, object)	Diferente de
>	__gt__(self, object)	Mayor que
>=	__ge__(self, object)	Mayor o igual que
<	__lt__(self, object)	Menor que
<=	__le__(self, object)	Menor o igual que
in	__contains__(self, value)	Operador de membresía
[index]	__getitem__(self, index)	Elemento en indice
len()	__len__(self)	Calcula número de elementos
str()	__str__(self)	Convierte objeto a string

Ver ejemplo: special_methods.py

Clases y Objetos: Herencia y Polimorfismo

- Sobreescribiendo la funcionalidad de los operadores:
  - Ejemplos:
    - special_methods.py
    - point.py

Clases y Objetos: Herencia y Polimorfismo

- Clases abstractas:
  - Las clases abstractas son clases que contienen uno o más métodos abstractos.
  - Un método abstracto es un método que se declara, pero que no contiene ninguna implementación.
  - Las clases abstractas no pueden ser instanciadas, y requieren subclases para proporcionar implementaciones para los métodos abstractos.
  - Aunque los métodos abstractos tengan implementación en la superclase, deben ser implementados en la subclase
  - Una clase que se deriva de una clase abstracta no puede ser instanciada a menos que todos sus métodos abstractos sean sobreescritos.
  - Un método abstracto puede tener una implementación en la clase abstracta! Incluso si se implementan, los diseñadores de subclases se verán obligados a sobreescribir la implementación.

Clases y Objetos: Herencia y Polimorfismo

- Clases abstractas:
  - Python viene con un módulo que proporciona la infraestructura para definir las Clases Base Abstractas (ABCs). Este módulo se llama - por razones obvias - abc.
  - El siguiente código Python utiliza el módulo abc y define una clase base abstracta:

```
from abc import ABC, abstractmethod
class AbstractClassExample(ABC):
    def __init__(self, value):
        self.value = value
        super().__init__()
    @abstractmethod
    def do_something(self):
        pass
```

Clases y Objetos: Herencia y Polimorfismo

- Clases abstractas:
  - Ejemplos:
    - Ver abstract_class_example_1.py
    - abstract_class_example_2.py
    - abstract_class_example_3.py

Clases y Objetos: Herencia y Polimorfismo

FIN

Programando con Python

Estructuras de datos

Estructuras de datos

- Son *estructuras* que mantienen algunos *datos* a la vez. En otras palabras, son usadas para almacenar una colección de datos relacionados.
- Hay cuatro estructuras de datos incorporadas en Python:
  - list
  - Tuple
  - set
  - diccionary

Estructuras de datos - List

- Es una estructura que mantiene una colección ordenada de elementos, por ejemplo, se puede almacenar una *secuencia* de elementos en una list
- Su formato viene dado por una secuencia de elementos separados por coma, entre corchetes:

  variable = [item1, item2, item3, …, itemN]

- Una vez creada una list, se pueden agregar, remover o buscar entre sus elementos. De allí que se diga que es un tipo de datos *mutable*

Estructuras de datos - List

- Métodos de la clase list:

Nombre(args)	Qué hace
list.**append**(x)	Agrega un elemento al final de la list
list.**extend**(*iterable*)	Extiende la list agregando todos los elementos del iterable
list.**insert**(*i*, x)	Inserta un elemento *i* en la posición x dada
list.**remove**(x)	Remueve el primer elemento de la list cuyo valor es x
list.**pop**([*i*])	Remueve y retorna el elemento *i* especificado. Si este no es declarado, remueve y retorna el último elemento de la list
list.**clear**()	Remueve todos los elementos de la list
list.**index**(x [, start[, end]])	Retorna un índice sobre base cero del primer elemento en la list cuyo valor es igual a x. Los argumentos opcionales *start* y *end* son interpretados como en la notación de un slide de la list y son usados para buscar una subsecuencia particular de la list
list.**count**(x)	Retorna el número de veces que x aparece en la list

Estructuras de datos - List

- Métodos de la clase list:

Nombre(args)	Qué hace
list.**sort**(*key = None, reverse = False*)	Ordena los elementos de la lista
list.**reverse**()	Invierte el orden de los elementos de la list
list.**copy**()	Retorna una copia de la list

- Funciones para trabajar con list:

Nombre(args)	Qué hace
len*(list)*	Retorna el número de elementos de la list
sum*(list)*	Retorna la suma de los elementos de la list
max*(list)*	Retorna el elemento de mayor valor en la list
min*(list)*	Retorna el elemento de menor valor en la list

Estructuras de datos - List

- Ejemplos:

```
>>> frutas = ['naranja', 'manzana', 'pera', 'cambur', 'kiwi', 'manzana', 'cambur']
>>> frutas.count('manzana')
2
>>> frutas.count('mandarina')
0
>>> frutas.index('cambur')
3
>>> frutas.index('cambur', 4)
6
```

Estructuras de datos - List

- Ejemplos:

```
>>> frutas.reverse()
>>> frutas
['cambur', 'manzana', 'kiwi', 'cambur', 'pera', 'manzana', 'naranja']
>>> frutas.append('fresa')
>>> frutas
['cambur', 'manzana', 'kiwi', 'cambur', 'pera', 'manzana', 'naranja', 'fresa']
>>> frutas.sort()
>>> frutas
['cambur', 'cambur', 'fresa', 'kiwi', 'manzana', 'manzana', 'naranja', 'pera']
>>> frutas.pop()
'pera'
```

Estructuras de datos - List

- Lists usadas como **stacks**:
  - Los métodos de list permiten usar una lista como un stack (pila), donde el último elemento es agregado es el primero retribuido (LIFO):

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

Estructuras de datos - List

- Lists usadas como **queues (colas)**:
  - También podemos implementar una queue donde el primer elemento ingresado es el primer elemento que sale (FIFO)
  - Si bien los métodos appends y pops se ejecutan rápido en una list, son ineficientes para crear la queue, ya que insertar o extraer del principio ocasiona que haya que correr todos los elementos de su posición.
  - Para implementar una queue, se usa **collections.deque**, la cual se diseñó para hacer appends y pops de cualquier extremo, de una forma rápida

Estructuras de datos - List

- Lists usadas como **queues (colas)**, ejemplo:

```
>>> from collections import deque
>>> queue = deque(["Juan", "María", "Alberto"])
>>> queue.append("Rosa")
>>> queue.append("José")
>>> queue.popleft()
'Juan'
>>> queue.popleft()
'María'
>>> queue
deque(['Alberto', 'Rosa', 'José']) ← mantiene el orden de llegada
>>>
```

Estructuras de datos - List

- Creando una list usando la función **range()**
  - La función range() retorna un objeto *iterable*
  - Simplemente se pasa el objeto a la list
  - Ejemplo1:

```
>>> list1 = list(range(8))
>>> list1
[0, 1, 2, 3, 4, 5, 6, 7]
```

Estructuras de datos - List

- Creando una list usando la función **range()**
  - Ejemplo2:

```
>>> list2 = list(range(20, 35))
>>> list2
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34]
```

  - Ejemplo 3:

```
>>> list3 = list(range(30, 60, 3))
>>> list3
[30, 33, 36, 39, 42, 45, 48, 51, 54, 57]
```

Estructuras de datos - List

- Las lists son **mutables**. Podemos modificarla sin crear un nuevo objeto
- Ejemplo:

```
>>> lista = ["carro", "pelota", "niño", "casa"]
>>> id(lista)
50947536
>>> lista[1] = "animal"
>>> lista
['carro', 'animal', 'niño', 'casa'] ← cambió el índice 1
>>> id(lista)
50947536 ← pero el objeto sigue siendo el mismo
```

Estructuras de datos - List

- Iterando los elemento de una list
- Se usan los bucles: for

```
>>> marcas = [10, 15, 33, 95, 44]
>>> for m in marcas:
...     print(m)
...
10
15
33
95
44
>>>
```

Estructuras de datos - List

- Iterando los elemento de una list
- Se usan los bucles: while

```
>>> marcas = [10, 15, 33, 95, 44]
>>> i = 0
>>> while i < len(marcas):
...     print(marcas[i])
...     i += 1
...
10
15
33
95
44
```

Estructuras de datos - List

- Al igual que con strings, con la list podemos:
  - Hacer **slicing**: lista[start: end]
  - Aplicar el operador **in** y **not in**
  - Concatenar: listC + listD
  - Aplicar el operador de repetición: listA = listB * 3

Estructuras de datos - List

- **Comprensión**: Cuando se desea crear una list donde cada elemento es el resultado de una operación, o cuando cumple con alguna condición
- Sintaxis: [expresión **for** elemento **in** iterable]
- Ejemplo: se desea obtener los cubos de una secuencia de 2 a 7:

```
>>> cubos = [ i ** 3 for i in range(1,8)]
>>> cubos
[1, 8, 27, 64, 125, 216, 343]
```

Estructuras de datos - List

- También se puede colocar una condición **if**
- Sintaxis: [expresión **for** elemento **in** iterable **if** condición]
- Ejemplo: se desea obtener los cubos de una secuencia de 2 a 7:

```
>>> cubos = [ i ** 3 for i in range(1,8) if i % 2 == 0]
>>> cubos
[8, 64, 216]
```

Estructuras de datos - Tuples

- Tuples
  - Forma parte de la familia de tipos de datos llamados secuencias, al igual que las **list** y el **range**
  - Al igual que las list, son una agrupación de datos separados por coma, pero encerrados entre paréntesis

```
>>>  t = 1, 3, 5, 7, 9
>>> t
>>>(1, 3, 5, 7, 9)
```

  - Son inmutables
  - Pero pueden contener múltiples objetos
  - Se usan en situaciones y propósitos diferentes de list

Estructuras de datos - Tuples

- Operaciones con Tuples
  - Se puede tener acceso a un elemento o hacer slicing con el operador **[]**
  - Las operaciones incluidas **min()**, **max()**, **sum()** son válidas
  - Los operadores de membresía **in** o **not in**
  - Los operadores de comparación
  - Los operadores **+** y *****
  - Hacer iteraciones con **for** o **while**

Estructuras de datos - Sets

- Sets
  - Es una colección de elementos sin orden que no permite elementos repetidos
  - Los datos se encuentran encerrados entre llaves "{ }"

    ```
    >>>  s = {1, 3, 5, 7, 9}
    ```

  - Son mutables, para agregar elementos se usa el método **add()**
  - Permiten eliminar elementos usando los métodos **remove()** y **discard()**. La diferencia radica en que discard() no genera error si el elemento no existe
  - No se puede tener acceso a sus elemento usando un índice

Estructuras de datos - Sets

- Sets – métodos (algunos):

Método	Descripción
add()	Agrega un elemento al set
clear()	Remueve todos los elementos del set
copy()	Retorna una copia del set
difference()	Retorna un set con la diferencia entre uno o más sets
discard()	Remueve el elemento especificado
intersection()	Retorna un set que es la intersección de otros dos sets
issubset()	Retorna un boolean indicando si otro set contiene a este
issuperset()	Retorna un boolean indicando si este set contiene a otro
pop()	Remueve el elemento indicado
update()	Actualiza el set con la unión de este y otros sets

Estructuras de datos - Dictionarios

- Dictionarios
  - Son conjuntos de datos agrupados en pares clave-valor
  - A diferencia de las secuencias los diccionarios son indexados por claves
  - La claves pueden ser de cualquier tipo inmutable: strings, números o tuples. SI un tuple contiene un objeto mutable, directo o indirecto, no puede ser usado como clave
  - Las claves deben ser únicas
  - Se usan {} para representarlos, de la manera *{clave: valor}*
  - Las principales operaciones de un dictionary es almacenar un valor con una clave y luego extraer el valor usando la clave.

Estructuras de datos - Dictionarios

- Dictionarios
  - Pueden ser cambiados, no son inmutables
  - Para eliminar un par *clave:valor* se usa **del**

```
>>> dic = {'juan': 1234, 'maria': 5678, 'rosa': 2468}
>>> del dic['maria']
>>> dic
{'juan': 1234, 'rosa': 2468}
```

  - **list(dic)**, retorna una lista de todas las claves del diccionario
  - El constructor **dict()**, crea diccionarios desde secuencias clave:valor

Estructuras de datos - Dictionaries

- Operaciones con dictionaries (algunas)

Método	Descripción
clear()	Remueve todos los elementos del diccionario
copy()	Retorna una copia del diccionario
fromkeys()	Retorna un diccionario con las claves y valores especificados
get()	Retorna el valor especificado por la clave
items()	Retorna una list conteniendo un tuple por cada clave:valor
keys()	Retorna una list con las claves del diccionario
pop()	Remueve el elemento con la clave especificada
popitem()	Remueve el ultimo par clave valor insertado
setdefault()	Retorna el valor de la clave especificada. SI esta no existe, inserta la clave con el valor indicado
update()	Actualiza el diccionario con la clave valor especificada
values()	Retorna una lista con todos los valores del diccionario

Técnicas de looping

- Al hacer un lazo a través de diccionarios, la clave y el correspondiente valor pueden ser obtenidos de una sola vez usando el método **items()**
- Al hacer un lazo en una secuencia, el índice de posición y su valor se pueden obtener usando la función **enumerate()**
- Para hacer loop sobre dos o mas secuencias al mismo tiempo, las entradas pueden se apareadas usando la función **zip()**
- Para hacer loop en forma inversa sobre una secuencia use la función **reversed()**

Técnicas de looping

- Para hacer loop sobre una secuencia de forma ordenada, use la función **sorted()**, la cual retorna una nueva lista ordenada, dejando la original intacta
- Si alguna vez se trata de cambiar una lista mientras está haciendo un loop sobre ella; es más simple y seguro hacer una copia de la lista.