

Introducción a la Programación para IA

Clase 1.2

Agenda

- Listas
- Diccionarios
- Conjuntos
- Composición condicional
- Composición iterativa
- Funciones
- Clases y objetos

Listas

- Es una de las colecciones de información más útiles y versátiles
- Corresponden a una organización lineal, indexable, de elementos
- Son mutables

```
fruits = ["apple", "orange", "tomato", "banana"] # a list of strings
print(type(fruits))
print(fruits)
```

```
<class 'list'>
['apple', 'orange', 'tomato', 'banana']
```

Indexación de una lista

- Indexación es el acceso a los elementos de una colección lineal por su posición (índice)

```
fruits[2]
```

```
'tomato'
```

- La indexación en listas se consigue con corchetes, y comienza en cero

Index:	0	1	2	3
List:	apple	orange	tomato	banana

Tamaño de una colección

Al acceder a elementos de una colección a través de índices es importante ser conscientes del tamaño de la colección

La función **len()** retorna el número de elementos en una lista.

```
len(fruits)
```

4

Las listas son mutables

Las listas son colecciones mutables. Admiten agregar y quitar elementos:

```
fruits.append("lime")    # add new item to list
print(fruits)
fruits.remove("orange")  # remove orange from list
print(fruits)
```

```
['apple', 'orange', 'apricot', 'banana', 'lime']
['apple', 'apricot', 'banana', 'lime']
```

Listas de valores numéricos

En Python es sumamente útil generar listas de valores numéricos, para diversas tareas.

La función ***range()*** genera secuencias de números en formato de lista:

```
nums = list(range(10))  
print(nums)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
nums = list(range(0, 100, 5))  
print(nums)
```

```
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85,  
90, 95]
```

Slicing de listas

- Slicing es el proceso de tomar un subconjunto de elementos de una lista (u otras colecciones indexables)
- Muy útil y flexible!

```
print(nums)
print(nums[1:5:2]) # Get from item 1(starting point) through item 5(end point, not included) with step size 2
print(nums[0:3]) # Get items 0 through 3(not included)
print(nums[4:]) # Get items 4 onwards
print(nums[-1]) # Get the last item
print(nums[::-1]) # Get the whole list backwards
```

```
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95]
[5, 15]
[0, 5, 10]
[20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95]
95
[95, 90, 85, 80, 75, 70, 65, 60, 55, 50, 45, 40, 35, 30, 25, 20, 15, 10, 5, 0]
```


Otras funciones sobre listas

- Las listas cuentan con una variedad amplia de funciones predefinidas

```
print(len(nums))    # number of items within the list
print(max(nums))    # the maximum value within the list
print(min(nums))    # the minimum value within the list
```

```
20
95
0
```

Listas heterogéneas

- Las listas admiten elementos de diferentes tipos:

```
mixed = [3, "Two", True, None]  
print(mixed)
```

```
[3, 'Two', True, None]
```

Tuplas

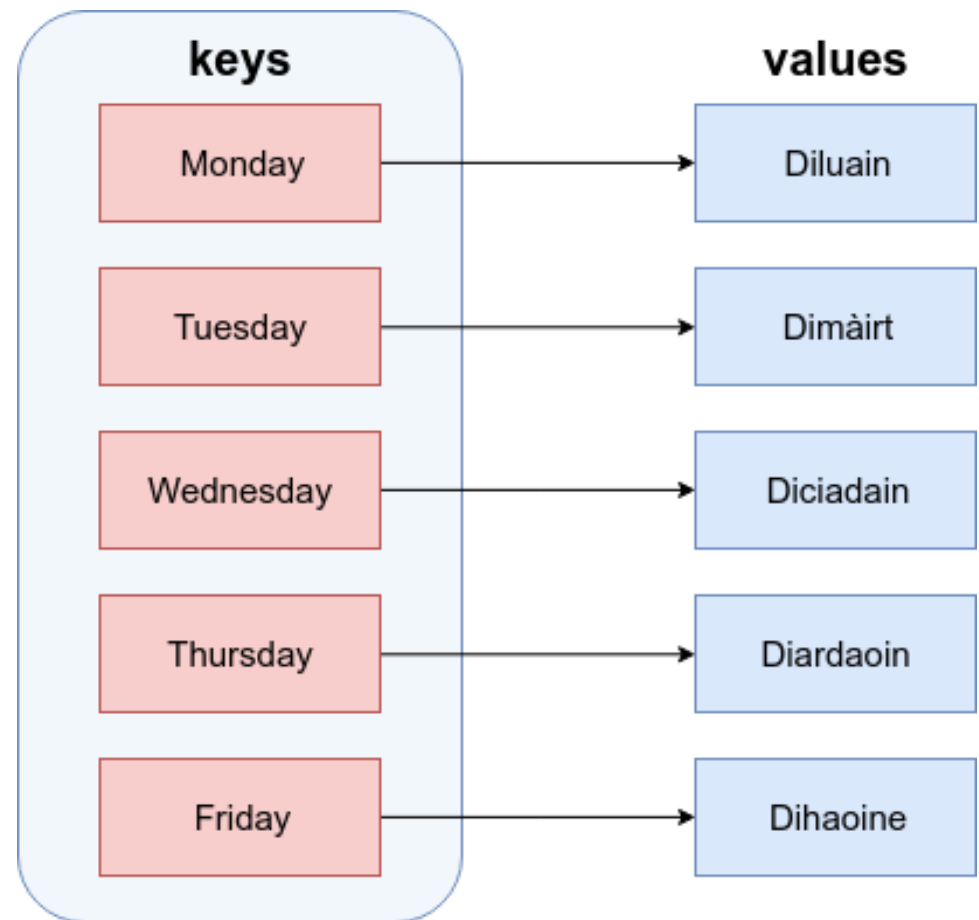
- Las tuplas también son secuencias - colecciones lineales de elementos
 - pero son inmutables
 - No se pueden cambiar (no se les puede agregar/quitar elementos)

```
fruits = ("apple", "orange", "tomato", "banana") # now the tomato is a fruit forever  
print(type(fruits))  
print(fruits)
```

```
<class 'tuple'>  
('apple', 'orange', 'tomato', 'banana')
```

Diccionarios

- Técnicamente, funciones parciales
 - Podemos pensarlas como conjuntos de pares ordenados
 - Cada elemento de un diccionario cuenta con una clave y un valor
 - Las claves se utilizan para acceder a los valores respectivos
 - Las claves son únicas en un diccionario (no pueden repetirse)



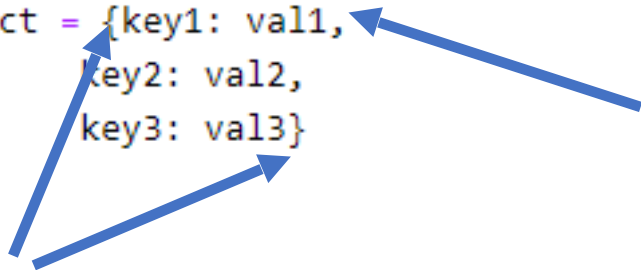
Definición de diccionarios por extensión

Podemos declarar/definir diccionarios por extensión, dando sus pares **clave : valor** :

```
mydict = {key1: val1,  
          key2: val2,  
          key3: val3}
```

Llaves

Los pares se separan con
comas

The diagram consists of two blue arrows. One arrow originates from the text 'Los pares se separan con comas' and points to the comma separating 'key1: val1' and 'key2: val2' in the dictionary definition. The second arrow originates from the text 'Llaves' and points to the 'key1' part of the first key-value pair.

Propiedades de los diccionarios

- Son mutables
- Hacen corresponder claves a valores
- Los valores se acceden a través de sus claves
- Las claves son únicas (e inmutables!)
- Los valores no pueden existir sin una clave correspondiente

Diccionarios

El siguiente es un ejemplo de definición de diccionario, con claves y valores de tipo string:

```
days = {"Monday": "Diluain", "Tuesday": "Dimàirt",  
        "Wednesday": "Diciadain", "Thursday": "Diardaoin",  
        "Friday": "Dihaoine"}  
print(type(days))  
print(days)
```

```
<class 'dict'>  
{'Monday': 'Diluain', 'Tuesday': 'Dimàirt', 'Wednesday': 'Diciadain',  
'Thursday': 'Diardaoin', 'Friday': 'Dihaoine'}
```

Acceso a valores de un diccionario

Los valores de un diccionario se acceden por sus respectivas claves, a través de la notación de corchetes:

```
days["Friday"]
```

```
'Dihaoine'
```

Los diccionarios no son colecciones indexables

Modificación de diccionarios

Los diccionarios ofrecen métodos para actualizar/modificar su contenido:

```
days.update({"Saturday": "Disathairne"})  
print(days)  
days.pop("Monday")  # Remove Monday because nobody likes it  
print(days)
```

```
{'Monday': 'Diluain', 'Tuesday': 'Dimàirt', 'Wednesday': 'Diciadain',  
'Thursday': 'Diardaoin', 'Friday': 'Dihaoine', 'Saturday': 'Disathairn  
e'}  
{'Tuesday': 'Dimàirt', 'Wednesday': 'Diciadain', 'Thursday': 'Diardaoi  
n', 'Friday': 'Dihaoine', 'Saturday': 'Disathairne'}
```

Claves y valores

Podemos obtener tanto las claves como los valores almacenados en un diccionario, perdiendo la correspondencia entre los mismos:

```
print(days.keys())    # get only the keys of the dictionary
print(days.values())  # get only the values of the dictionary

dict_keys(['Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday'])
dict_values(['Dimàirt', 'Diciadain', 'Diardaoin', 'Dihaoine', 'Disathairne'])
```

Conjuntos

- Colecciones de elementos, sin duplicados, y no indexadas
- No soportan operaciones de indexación, ni siquiera slicing
- Se definen con llaves {}, o se pueden crear a través de funciones que los retornan (o convierten a partir de otros tipos):

```
x = set([1, 2, 3]) # a set created from a list
print(type(x))
print(x)
y = {1, 2, 3}      # a set created directly

x == y             # x and y are the same object
```

```
<class 'set'>
{1, 2, 3}
```

```
True
```

Composición iterativa

- Construcción fundamental de programas (ya vimos asignación y composición secuencial!)

The diagram illustrates the execution of an if-else statement. It consists of the following code and annotations:

```
if answer:  
    close program  
else:  
    continue running program
```

Annotations with arrows pointing to the code:

- An arrow points from the text "Condición" to the `answer:` part of the `if` statement.
- An arrow points from the text "Se ejecuta si la condición es True" to the `close program` line.
- An arrow points from the text "Se ejecuta si la condición es False" to the `continue running program` line.

The code is formatted with `if` and `else:` in green, and the body lines are indented. The body lines `close program` and `continue running program` are enclosed in orange boxes.

Ejemplo

La indentación es fundamental en la definición de bloques!

```
x = True
if x:
    print("Executing if")
else:
    print("Executing else")
print("Prints regardless of the outcome of the if-else block")
```

Executing if

Prints regardless of the outcome of the if-else block

Sobre la indentación

- La indentación define los bloques
- Como en otros lenguajes, conviene ser consistentes en la indentación.
 - En general, 4 espacios.

```
x = 10
if x%2 == 0:
    print(x, 'is even!')
    if x%5 == 0:
        print(x, 'is divisible by 5!')
        print('Output only when x is divisible by both 2 and 5.')
    else:
        print(x, 'is not divisible by 5!')
        print('Output only when x is divisible by 2 but not divisible by 5.')
else:
    print(x, 'is odd!')
print('No indentation. Output in all cases.')
```

10 is even!
10 is divisible by 5!
Output only when x is divisible by both 2 and 5.
No indentation. Output in all cases.

Cadenas de condiciones (else if)

- Podemos encadenar condiciones mediante **elif**

```
if condition1:  
    condition 1 was True  
elif condition2:  
    condition 2 was True  
else:  
    neither condition 1 or condition 2 were True
```

- elif = else + if, indica que si las condiciones anteriores resultaron falsas, comprobaremos una nueva condición

Composición iterativa - Ciclos for

- La composición iterativa permite repetir un bloque de código
- En la sentencia for, iteramos sobre un conjunto de valores de una estructura de datos
- La indentación es muy importante, como en cualquier composición o anidamiento de bloques

```
for item in itemList:  
    do something to item
```


Ejemplo

```
fruits = ["apple", "orange", "tomato", "banana"]  
print("The fruit", fruits[0], "has index", fruits.index(fruits[0]))  
print("The fruit", fruits[1], "has index", fruits.index(fruits[1]))  
print("The fruit", fruits[2], "has index", fruits.index(fruits[2]))  
print("The fruit", fruits[3], "has index", fruits.index(fruits[3]))
```

```
The fruit apple has index 0  
The fruit orange has index 1  
The fruit tomato has index 2  
The fruit banana has index 3
```

Ejemplo

```
fruitList = ["apple", "orange", "tomato", "banana"]  
for fruit in fruitList:  
    print("The fruit", fruit, "has index", fruitList.index(fruit))
```

```
The fruit apple has index 0  
The fruit orange has index 1  
The fruit tomato has index 2  
The fruit banana has index 3
```

Ciclo for con valores numéricos

```
numbers = list(range(10))  
for num in numbers:  
    squared = num ** 2  
    print(num, "squared is", squared)
```

```
0 squared is 0  
1 squared is 1  
2 squared is 4  
3 squared is 9  
4 squared is 16  
5 squared is 25  
6 squared is 36  
7 squared is 49  
8 squared is 64  
9 squared is 81
```

Composición iterativa - Ciclos While

- Es la forma más general de composición iterativa.
- Se itera hasta que la condición del ciclo se haga falsa
 - Debe tenerse particular cuidado en no caer en ciclos infinitos

```
n = 0
while n < 5:
    print("Executing while loop")
    n = n + 1

print("Finished while loop")
```

```
Executing while loop
Executing while loop
Executing while loop
Executing while loop
Executing while loop
Finished while loop
```

Funciones

- Elemento fundamental para abstracción procedimental
- Permite asignar un nombre a una funcionalidad específica
 - Habilita la modularidad
 - Mejora la comprensión de código
 - Permite el reuso de funcionalidades

Declaración de funciones

Palabra clave

Cualquier número de argumentos

```
def functionName(argument1, argument2, argument3, ... argumentN):  
    statements..  
    ..  
    ..  
  
    return returnValue
```

[Opcional] Sale de la función y retorna un valor

- Las funciones aceptan argumentos y ejecutan un bloque de código (el cuerpo de la función)
- Pueden retornar valores
 - Cuando no lo hacen, son comandos o procedimientos

Ejemplo de una función

```
def printNum(num):  
    print("My favourite number is", num)  
  
printNum(7)  
printNum(14)  
printNum(2)
```

My favourite number is 7
My favourite number is 14
My favourite number is 2

Otro ejemplo

```
x = 3.4
remainder = x % 1
if remainder < 0.5:
    print("Number rounded down")
    x = x - remainder
else:
    print("Number rounded up")
    x = x + (1 - remainder)

print("Final answer is", x)
```

Number rounded down
Final answer is 3.0

Otro ejemplo (cont.):

```
def roundNum(num):  
    remainder = num % 1  
    if remainder < 0.5:  
        return num - remainder  
    else:  
        return num + (1 - remainder)  
  
# Will it work?  
x = roundNum(3.4)  
print (x)  
  
y = roundNum(7.7)  
print(y)  
  
z = roundNum(9.2)  
print(z)
```

3.0
8.0
9.0

Un último ejemplo de función

$$(val - src[0]) \times \frac{dst[1] - dst[0]}{src[1] - src[0]} + dst[0]$$

```
# Generic scale function
# Scales from src range to dst range
def scale(val, src, dst=(-1,1)):
    return (int(val - src[0]) / (src[1] - src[0])) * (dst[1] - dst[0]) + dst[0]

print(scale(49, (-100,100), (-50,50)))
print(scale(49, (-100,100)))
```

24.5

0.49

Clases

- Constituyen un elemento fundamental en la organización de aplicaciones
 - Habilitan la programación orientada a objetos
- Las clases definen módulos y simultáneamente nuevas abstracciones de datos
- Encapsulan datos con las operaciones que los acceden
 - Datos: campos o atributos
 - Operaciones: métodos

Declaración de Clases en Python

- Declaración de una clase:

```
class name:  
    statements
```

Campos o atributos

name = value

– Ejemplo:

```
class Point:
    x = 0
    y = 0

# main
p1 = Point()
p1.x = 2
p1.y = -5
```

point.py

```
1 class Point:
2     x = 0
3     y = 0
```

- La declaración de variables dentro de una clase define atributos de clase (estáticos)
 - Compartidos por todos los objetos de la clase
- Python no cuenta con construcciones para encapsulamiento, ocultamiento o visibilidad

Uso de clases

- Las clases definen templates para la creación de objetos
- El uso de clases requiere:
 - En general, la importación de los archivos que declaran la o las clases
 - La generación de objetos, o instancias de clase

point_main.py

```
1  from Point import *
2
3  # main
4  p1 = Point()
5  p1.x = 7
6  p1.y = -3
7  ...
8
9  # Python objects are dynamic (can add fields any time!)
10 p1.name = "Tyler Durden"
```

Métodos

- Los métodos son funciones declaradas en el contexto de clases.
- Definen funcionalidades disponibles en todas las instancias de una clase (objetos)

```
def name(self, parameter, ..., parameter) :  
    statements
```

- `self` debe ser el primer parámetro de los métodos (de objeto, y representa el objeto al cual se aplica el método)
 - (Similar a `this` en Java)
- El acceso a campos o métodos de objeto se realiza utilizando la referencia a `self`

```
class Point:  
    def translate(self, dx, dy):  
        self.x += dx  
        self.y += dy  
    ...
```

Ejemplo

point.py

```
1 from math import *
2
3 class Point:
4     x = 0
5     y = 0
6
7     def set_location(self, x, y):
8         self.x = x
9         self.y = y
10
11     def distance_from_origin(self):
12         return sqrt(self.x * self.x + self.y * self.y)
13
14     def distance(self, other):
15         dx = self.x - other.x
16         dy = self.y - other.y
17         return sqrt(dx * dx + dy * dy)
```


Constructores

- Los constructores son métodos especiales.
- Tienen, en general, la finalidad de inicializar el estado de los objetos de una clase
 - En Python tienen una responsabilidad adicional: declarar los campos/atributos de objetos de la clase

```
def __init__(self, parameter, ..., parameter) :  
    statements
```

– Ejemplo:

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

Otros métodos especiales

- Python tiene algunos otros métodos especiales, por ejemplo, para sobrecargar operadores, o facilitar impresión
- Uno de estos es el método `__str__`
 - Es el método que se invoca cuando se “imprime” un objeto de la clase correspondiente

```
def __str__(self):  
    return "(" + str(self.x) + ", " + str(self.y) + ")"
```

Ejemplo

point.py

```
1 from math import *
2
3 class Point:
4     def __init__(self, x, y):
5         self.x = x
6         self.y = y
7
8     def distance_from_origin(self):
9         return sqrt(self.x * self.x + self.y * self.y)
10
11    def distance(self, other):
12        dx = self.x - other.x
13        dy = self.y - other.y
14        return sqrt(dx * dx + dy * dy)
15
16    def translate(self, dx, dy):
17        self.x += dx
18        self.y += dy
19
20    def __str__(self):
21        return "(" + str(self.x) + ", " + str(self.y) + ")"
```