Introducción a la Inteligencia Artificial

Clase 1.2. Colecciones. Estructuras de control, funciones y clases (en Python)

Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, 2024

Agenda

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

Listas

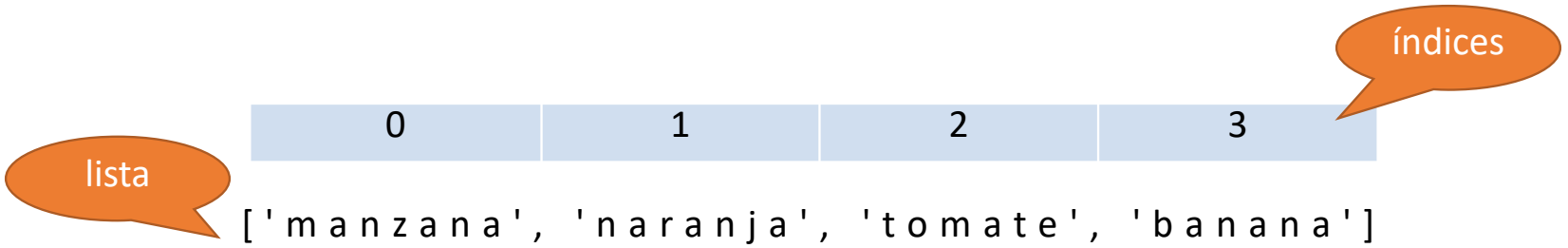
- Es una de las colecciones de información más útiles y versátiles
- Corresponden a una organización lineal, indexable, de elementos
- Son mutables

Indexación de una lista

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

```
In [4]: frutas[2]
Out[4]: 'tomate'
```

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



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.

- Intentar acceder al elemento en una posición inválida (e.g., mayor o igual a la longitud de la lista), dará lugar a un error (excepción) de índice fuera de rango

```
In [5]: len(frutas)
Out[5]: 4
```

Mutabilidad de listas

Las listas son colecciones mutables. Admiten agregar y quitar elementos

- pop y remove permitan quitar elementos, por índice y valor respectivamente
- append e insert permiten agregar elementos

```
In [18]: frutas = ["manzana", "naranja", "tomate", "banana"]

In [19]: frutas.append('lima')

In [20]: frutas.insert(2, 'durazno')

In [21]: frutas
Out[21]: ['manzana', 'naranja', 'durazno', 'tomate', 'banana', 'lima']

In [22]: frutas.pop(1)
Out[22]: 'naranja'

In [23]: frutas.remove('tomate')

In [24]: frutas
Out[24]: ['manzana', 'durazno', 'banana', 'lima']
```

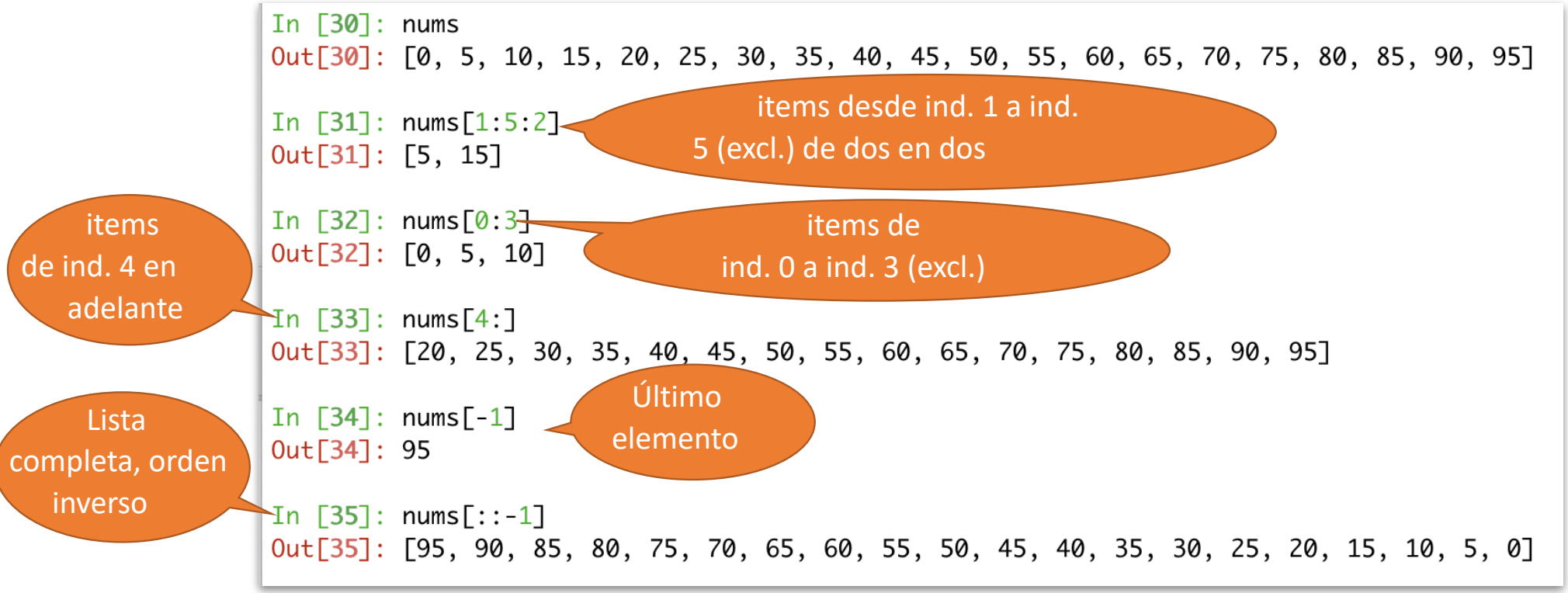
Listas de valores numéricos

En Python es sumamente útil generar listas de valores numéricos, para diversas tareas.
La función ***range()*** permite generar secuencias de números en formato de lista (via casts):

```
In [27]: nums = list(range(10))

In [28]: nums
Out[28]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [29]: nums = list(range(0, 100, 5))

In [30]: nums
Out[30]: [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!



Otras funciones sobre listas

- Las listas cuentan con una variedad amplia de funciones predefinidas

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

In [41]: min(nums)
Out[41]: 0

In [42]: max(nums)
Out[42]: 95

In [43]: min(nums)
Out[43]: 0
```

Listas heterogéneas

- En Python, las listas pueden ser heterogéneas, es decir, admitir valores de diferentes tipos
  - En general, sin embargo, trabajaremos con listas homogéneas, o al menos donde los tipos de los elementos tengan algún aspecto común

```
In [47]: mixed = [3, "Dos", True, None]

In [48]: mixed
Out[48]: [3, 'Dos', True, None]
```

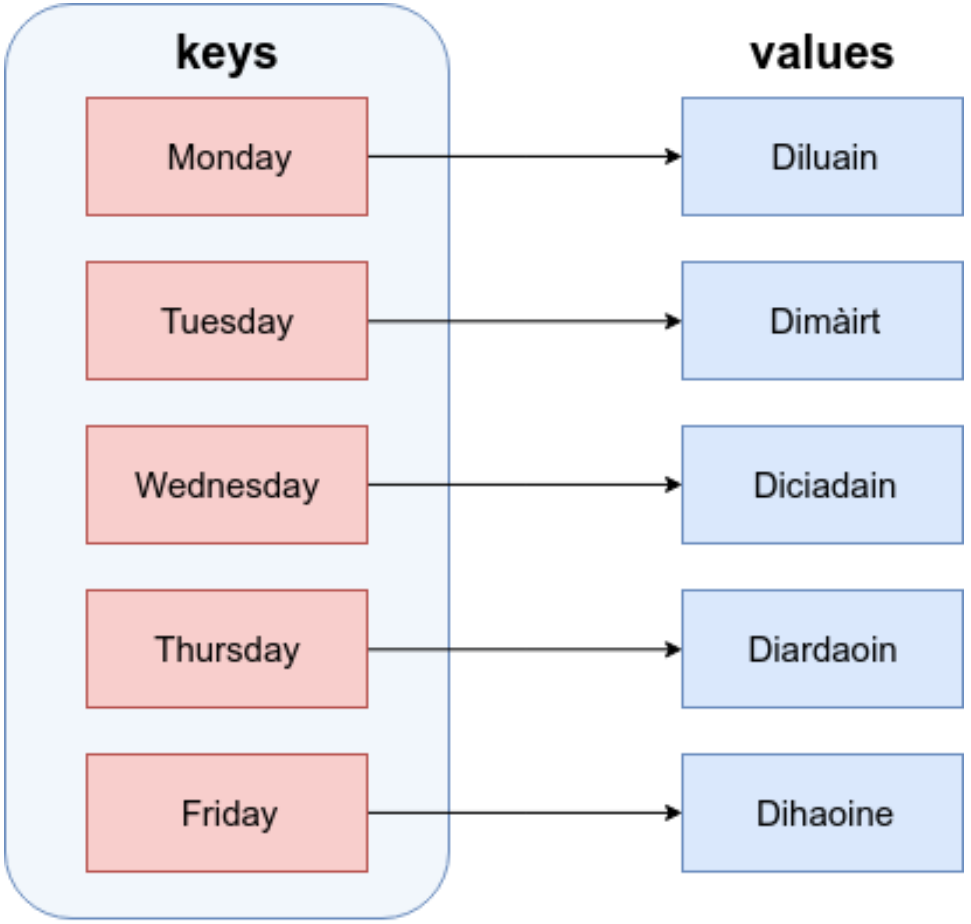
Tuplas

- Las tuplas también son secuencias - colecciones lineales de elementos - pero son inmutables
  - No se pueden cambiar (no se les puede agregar/quitar elementos)
  - Admiten acceso por indexación, con una notación similar a la de listas

```
In [49]: frutas = ("manzana", "naranja", "tomate", "banana")

In [50]: type(frutas)
Out[50]: tuple

In [51]: frutas
Out[51]: ('manzana', 'naranja', 'tomate', 'banana')
```

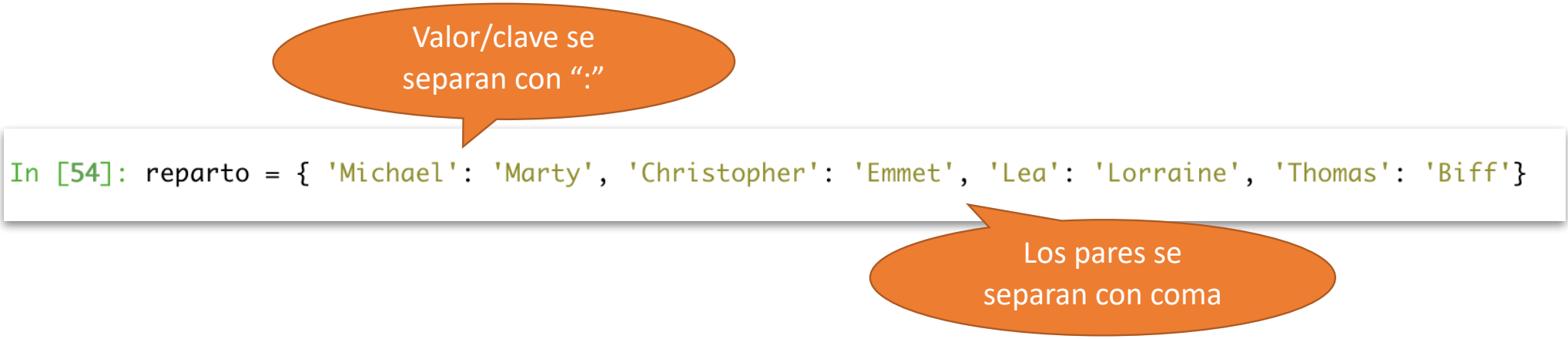
Diccionarios

- 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** :



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:

```
In [54]: reparto = { 'Michael': 'Marty', 'Christopher': 'Emmet', 'Lea': 'Lorraine', 'Thomas': 'Biff'}

In [55]: type(reparto)
Out[55]: dict

In [56]: reparto
Out[56]:
{'Michael': 'Marty',
 'Christopher': 'Emmet',
 'Lea': 'Lorraine',
 'Thomas': 'Biff'}

In [57]: reparto['Lea']
Out[57]: 'Lorraine'
```

Acceso a valores de un diccionario

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

```
In [57]: reparto['Lea']
Out[57]: 'Lorraine'
```

Los diccionarios no son colecciones indexables

Modificación de diccionarios

Los diccionarios ofrecen métodos para actualizar/ modificar su contenido:

```
In [66]: reparto
Out[66]:
{'Michael': 'Marty',
 'Christopher': 'Emmet',
 'Lea': 'Lorraine',
 'Thomas': 'Biff'}

In [67]: reparto.update({'Christopher': 'Doc'})

In [68]: reparto.update({'Claudia': 'Jennifer'})

In [69]: reparto.update({'Elizabeth': 'Jennifer'})

In [70]: reparto.pop('Claudia')
Out[70]: 'Jennifer'

In [71]: reparto
Out[71]:
{'Michael': 'Marty',
 'Christopher': 'Doc',
 'Lea': 'Lorraine',
 'Thomas': 'Biff',
 'Elizabeth': 'Jennifer'}
```

Claves y valores

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

```
In [73]: reparto
Out[73]:
{'Michael': 'Marty',
 'Christopher': 'Doc',
 'Lea': 'Lorraine',
 'Thomas': 'Biff',
 'Elizabeth': 'Jennifer'}

In [74]: reparto.keys()
Out[74]: dict_keys(['Michael', 'Christopher', 'Lea', 'Thomas', 'Elizabeth'])

In [75]: reparto.values()
Out[75]: dict_values(['Marty', 'Doc', 'Lorraine', 'Biff', 'Jennifer'])
```

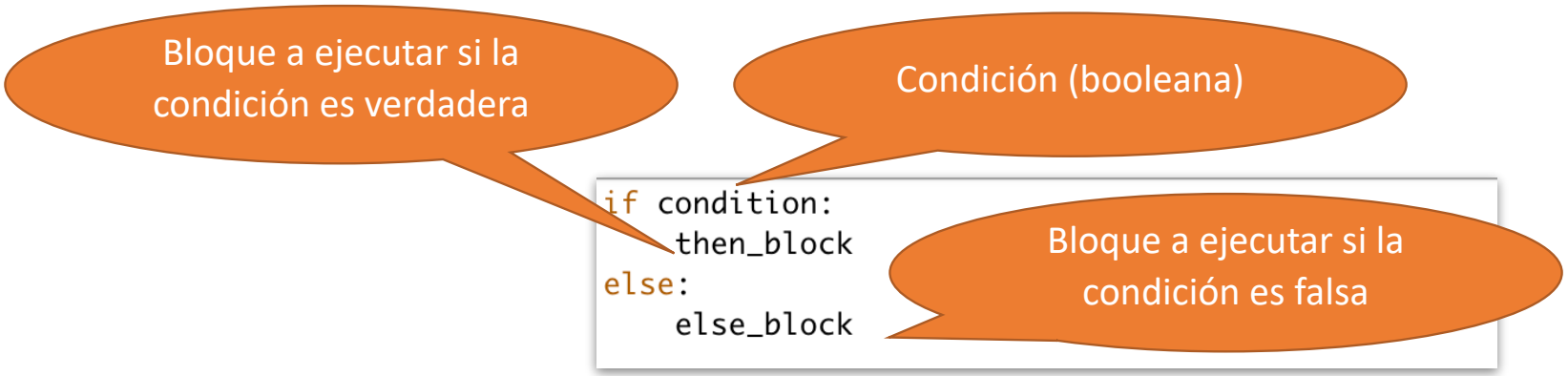
Conjuntos

- 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).

```
In [94]: primos = set()

In [95]: primos.update({2})

In [96]: primos.update({3})

In [97]: primos.update({5, 7})

In [98]: primos
Out[98]: {2, 3, 5, 7}

In [99]: primos.update({9, 11})

In [100]: primos.remove(9)

In [101]: primos
Out[101]: {2, 3, 5, 7, 11}
```

Composición Condicional

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



Ejemplo

En Python, el nivel de indentación define los bloques

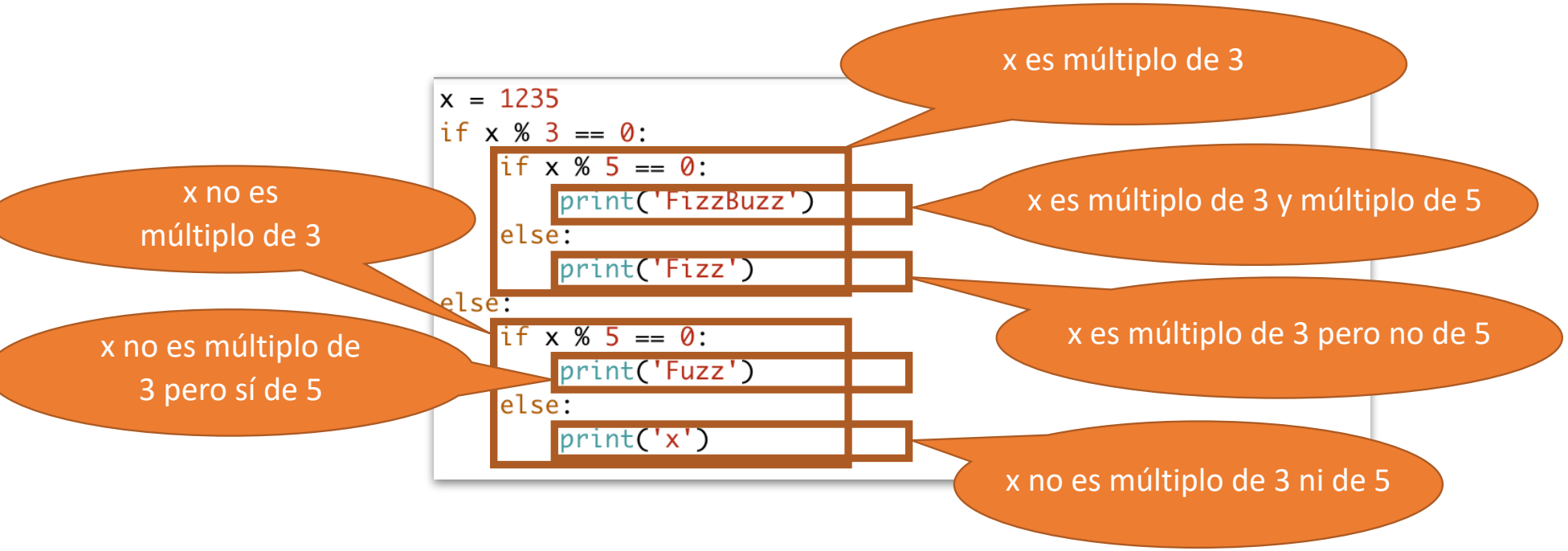
indentación es fundamental en el comportamiento de programas!

```
x = -1235
if x > 0:
    print(f'el valor absoluto de {x} es {x}')
else:
    print(f'el valor absoluto de {x} es {-x}')
print('listo')
```

```
(base) aguirre@192 practicas % python ejemplo.py
el valor absoluto de -1235 es 1235
listo
(base) aguirre@192 practicas %
```

Sobre la indentación

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



Cadenas de condiciones (else if)

- Podemos encadenar condiciones mediante **elif**

```
x = 1235
if x % 3 == 0 and x % 5 == 0:
    print('FizzBuzz')
elif x % 3 == 0:
    print('Fizz')
elif x % 5 == 0:
    print('Fuzz')
else:
    print('x')
```

¿?

- 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 collection:
    do_something
```

Ejemplo

```python
for x in list(range(1,101)):
    if x % 3 == 0 and x % 5 == 0:
        print('FizzBuzz')
    elif x % 3 == 0:
        print('Fizz')
    elif x % 5 == 0:
        print('Fuzz')
    else:
        print(x)
```

```
1
2
Fizz
4
Fuzz
Fizz
7
8
Fizz
Fuzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Fuzz
Fizz
22
23
Fizz
...
```

Composición iterativa - Ciclos While

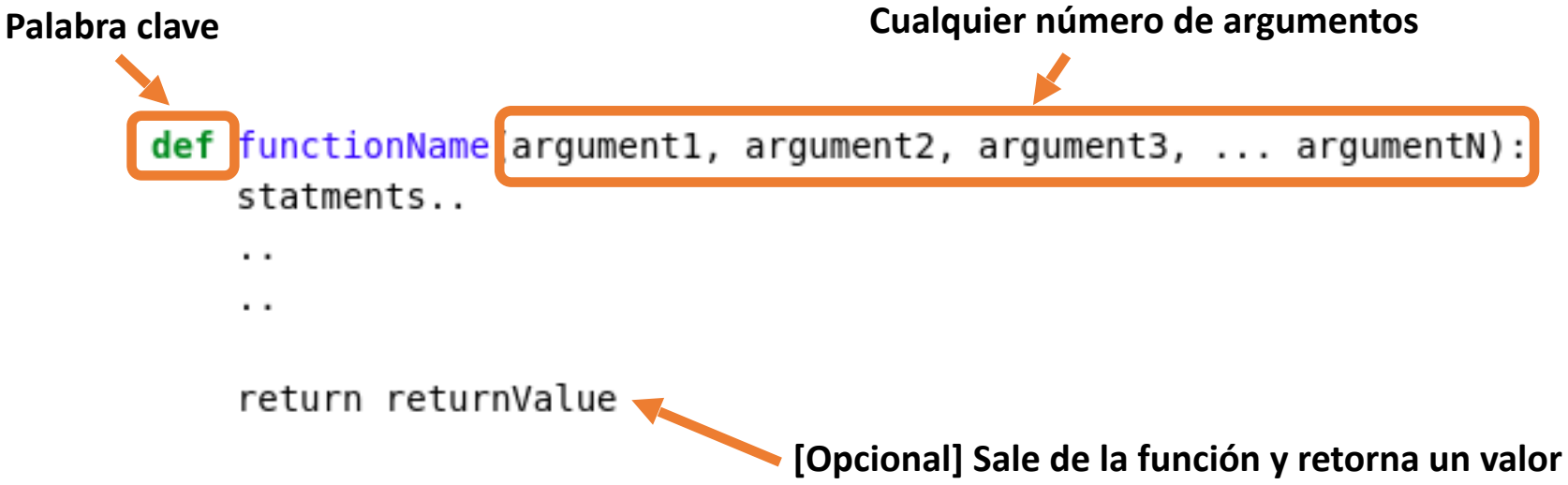
- Es la forma más general de composición iterativa.
- Se itera hasta que la condición del ciclo se haga falsa
  - Debe tenerse particular cuidado en no caer en ciclos infinitos

```
n = 1
while n <= 100:
    if n % 3 == 0 and n % 5 == 0:
        print('FizzBuzz')
    elif n % 3 == 0:
        print('Fizz')
    elif n % 5 == 0:
        print('Fuzz')
    else:
        print(n)
    n = n + 1
```

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



- Las funciones aceptar 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 fizzbuzz(n):
    if n % 3 == 0 and n % 5 == 0:
        return 'FizzBuzz'
    elif n % 3 == 0:
        return 'Fizz'
    elif n % 5 == 0:
        return 'Fuzz'
    else:
        return n

x = 1
while x <= 100:
    print(fizzbuzz(x))
    x = x + 1
```

```
(base) aguirre@192 practicas % python ejemplo.py
1
2
Fizz
4
Fuzz
Fizz
7
8
Fizz
Fuzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Fuzz
Fizz
22
...
```

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.):

```
import random

def minmax(ls):
    if len(ls) == 0:
        return (None, None)
    else:
        result = (ls[0], ls[0])
        for i in list(range(1, len(ls))):
            if ls[i] < result[0]:
                result = (ls[i], result[1])
            if ls[i] > result[1]:
                result = (result[0], ls[i])
        return result

my_list = list(range(0, 1000))
random.shuffle(my_list)
print(my_list)
print(minmax(my_list))
```

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) + ")"
```