

# Python Academy

# Fundamentals

Clase 3

¡Bienvenidos!



# Temario

- Estructuras de datos
  - Tuplas
  - Conjuntos (sets)
  - Diccionarios
  - Técnicas de bucle
- Generadores
- Leyendo y escribiendo archivos
  - La función **open()**
  - La sentencia **with**
- Clases
  - Cómo se definen
  - Objetos de clase e instancias
  - Métodos
- Módulos
  - Cómo se importan
  - Creando nuestros propios módulos
  - **`__init__.py`**



# Estructuras de datos

- Tuplas:

```
# una tupla consiste de valores separados por coma
tupla_1 = 'python', 3, 'iClase 3!'

print(tupla_1)    # ('python', 3, 'iClase 3!')

# y pueden estar anidadas

tupla_2 = tupla_1, (1, 2, 3, 4, 5)

print(tupla_2)    # (('python', 3, 'iClase 3!'), (1, 2, 3, 4, 5))

# son inmutables

tupla_1[0] = 'ruby'      # TypeError: 'tuple' object does not support item assignment

# pero pueden contener objetos mutables

tupla_3 = ([ 'python', 'academy'], 'fundamentals')

tupla_3[0][0] = 'python3'

print(tupla_3)    # ([ 'python3', 'academy'], 'fundamentals')
```

```
"""
Los paréntesis no son propios de las tuplas, pero se
muestran para diferenciarlas.
"""

nada = ()

algo = 'python',      # equivalente a decir ('python',)

len(nada)    # 0

len(algo)    # 1
```

```
"""
Tienen dos métodos iguales a las listas.
"""

tupla = (1, 2, 3, 4, 5)

tupla.count(1)    # 1

tupla.index(5)    # 4
```



# Estructuras de datos

- Conjuntos:

```
# los conjuntos no admiten valores repetidos
canasta = {'manzana', 'naranja', 'banana', 'uva', 'pera', 'manzana', 'uva', 'pera'}
print(canasta)    # {'banana', 'naranja', 'uva', 'manzana', 'pera'}
```

# el método set() admite sólo un argumento, que es tratado como una colección

```
a = set('abracadabra')
b = set('alacazam')

print(a)    # {'a', 'b', 'd', 'c', 'r'} --> letras únicas en <a>
print(b)    # {'l', 'z', 'm', 'a', 'c'} --> letras únicas en <b>

# operaciones con conjuntos

print(a - b)    # {'b', 'r', 'd'} --> letras en <a> pero no en <b>
print(a | b)    # {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'} --> letras en <a>, <b> o ambas
print(a & b)    # {'a', 'c'} --> letras en común entre <a> y <b>
print(a ^ b)    # {'r', 'd', 'b', 'm', 'z', 'l'} --> letras en <a> y <b>, pero no en ambas
```

```
"""
Para crear un conjunto vacío se puede usar
set() pero no {}.
"""

type(set())    # set

type({})        # dict

"""
Admiten ser generados por comprensión.
"""

a = {x for x in 'alacazam' if x not in 'abc'}
print(a)        # {'l', 'z', 'm'}
```



# Estructuras de datos

- Algunos métodos:

```
a = set(range(5))      # {0, 1, 2, 3, 4}

b = set(range(0, 5, 2))    # {0, 2, 4}

"""
set.add
"""

a.add(5)      # {2, 4, 5, 3, 0, 1}

"""

set.remove
"""

a.remove(5)    # {3, 1, 4, 2, 0}

"""

set.pop
"""

a.pop()       # 2

print(a)      # {0, 4, 1, 3}
```

```
"""
set.difference
"""

a.difference(b)    # {3, 0, 1} --> no lo cambia

"""
set.difference_update
"""

a.difference_update(b)

print(a)    # {0, 3, 1} --> lo actualiza

"""
set.issubset
"""

a.issubset(b)    # False --> si <b> contiene a <a>

"""
set.issuperset
"""

a.issuperset(b)   # False --> si <a> contiene a <b>
```

```
"""
set.union
"""

a.union(b)      # {4, 2, 1, 3, 0}

"""
set.update
"""

a.update(b)     # actualiza <a> con <b>

print(a)      # {3, 4, 0, 2, 1}

"""
set.copy
"""

c = a.copy()

print(c)      # {2, 0, 4, 3, 1}

"""
set.clear
"""

a.clear()

print(a)      # set()
```



# Estructuras de datos

- o Diccionarios:

```
"""
Los diccionarios están compuestos por claves (keys) y valores (values).

La asociación se delimita con ':' y los pares de claves/valores con ','.

agenda = {'Ana': 43782348, 'Beto': 45872394, 'Charly': 47893489}

agenda['Ana']      # 43782348

agenda['Dani'] = 42875937

print(agenda)      # {'Ana': 43782348, ... 'Dani': 42875937}
```

```
"""
Consultar su contenido como si fueran cualquiera de las demás
estructuras sólo hace referencia a sus claves.

"""
```

```
list(agenda)      # ['Ana', 'Beto', 'Charly', 'Dani']

al_reves = sorted(agenda, reverse=True)

print(al_reves)      # ['Dani', 'Charly', 'Beto', 'Ana']

'Ana' in agenda      # True

'Juan' in agenda      # False
```

```
"""
El constructor dict() crea diccionarios a partir de secuencias
de claves y valores.

"""

dict([('Ana', 43782348),
       ('Beto', 45872394),
       ('Charly', 47893489),
       ('Dani', 42875937)])
```

```
"""
Admiten ser generados por comprensión.

"""
```

```
cuadrados = {x: x**2 for x in range(2, 6)}

print(cuadrados)      # {2: 4, 3: 9, 4: 16, 5: 25}
```



# Estructuras de datos

- Algunos métodos:

```
numeros = dict(uno=1, dos=2, tres=3, cuatro=4)

print(numeros)      # {'uno': 1, 'dos': 2, 'tres': 3, 'cuatro': 4}

"""

dict.get

"""

numeros.get('uno')      # 1 --> lo mismo que numeros['uno']

"""

dict.copy

"""

copia = numeros.copy()

print(copia)      # {'uno': 1, 'dos': 2, 'tres': 3, 'cuatro': 4}

"""

dict.pop

"""

copia.pop('cuatro')      # 4

print(copia)      # {'uno': 1, 'dos': 2, 'tres': 3}
```

```
"""
dict.keys
"""

keys = numeros.keys()      # dict_keys(['uno', ... 'cuatro'])

type(keys)      # dict_keys

"""

dict.values
"""

numeros.values()      # dict_values([1, 2, 3, 4])

"""

dict.items
"""

numeros.items()      # dict_items([('uno', 1), ... ('cuatro', 4)])

"""

dict.update
"""

numeros.update({'cinco': 5})

numeros.update(seis=6)

print(numeros)      # {'uno': 1, ... 'cinco': 5, 'seis': 6}
```



# Estructuras de datos

- Algunas técnicas de bucle:

```
# accedemos a las claves y valores de un diccionario con dict.items()  
  
personajes = {'John': 'Wick', 'Thomas': 'Anderson', 'Kevin': 'Lomax'}  
  
for nombre, apellido in personajes.items():  
  
    print(nombre, apellido)  
  
# John Wick  
# Thomas Anderson  
# Kevin Lomax
```

```
# usamos set() para eliminar duplicados y sorted() para ordenar  
  
canasta = ['uva', 'banana', 'kiwi', 'naranja',  
           'uva', 'manzana', 'kiwi', 'naranja']  
  
for fruta in sorted(set(canasta)):  
  
    print(fruta)  
  
# banana  
# kiwi  
# manzana  
# naranja  
# uva
```

```
# iteramos sobre dos (o más) secuencias con zip(sec1, sec2, ...)  
  
objetos = ['un anillo', 'siete esferas', 'una masterball']  
  
propositos = ['gobernarlos a todos', 'revivir a Goku', 'atrapar a Mewtwo']  
  
for obj, prop in zip(objetos, propositos):  
  
    print(f'Necesito {obj} para {prop}.')  
  
# Necesito un anillo para gobernarlos a todos.  
# Necesito siete esferas para revivir a Goku.  
# Necesito una masterball para atrapar a Mewtwo.
```

```
# damos vuelta una secuencia con reversed()  
  
for impar in reversed(range(1, 10, 2)):  
  
    print(impar)  
  
# 9  
# 7  
# 5  
# 3  
# 1
```



# Generadores

- La sentencia **yield**:

```
# los generadores se diferencian de funciones convencionales
# por el uso de la sentencia 'yield'

def fibonacci(n):
    """
    Devuelve números de la serie Fibonacci
    menores que <n>.
    """
    a, b = 0, 1

    while a < n:
        yield a

        a, b = b, a+b

fib = fibonacci(50)      # iniciamos un generador hasta 50

for n in fib:
    print(n, end=' ')
# 0 1 1 2 3 5 8 13 21 34
```

```
# usamos next() para conseguir el siguiente valor en la secuencia

fib = fibonacci(10)      # iniciamos un generador hasta 10

next(fib)    # 0
next(fib)    # 1
next(fib)    # 1
next(fib)    # 2
next(fib)    # 3
next(fib)    # 5
next(fib)    # 8
next(fib)    # resulta en una excepción del tipo StopIteration
```

```
# podemos generar una lista por comprensión usando un generador

fib = [n for n in fibonacci(100)]

print(fib)    # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```



# Leyendo y escribiendo archivos

- La función `open()`:

```
"""
open() devuelve un objeto de tipo archivo, y es comunmente utilizado
con dos argumentos.
"""

# nombre del archivo
f = open('/ruta/al/archivo.txt', 'r') # modo de operación

f.readline()    # devuelve el primer renglón
f.readline()    # devuelve el segundo renglón

f.close()       # cierra el archivo

f = open('archivo.txt', 'a') # 'a' = 'append' --> escribe al final
f.write('nuevo renglón\n')
f.close()

f = open('archivo.txt', 'w') # 'w' borra los contenidos del archivo
cont = 'este es\nel nuevo contenido\n'
f.write(cont)
f.close()
```

```
"""
Los bloques try ... finally son particularmente convenientes
para trabajar con archivos.
"""

try:

    a = open('archivo.txt', 'r+') # 'r+' = lectura y escritura

    renglones = ['1º renglón\n',
                 '2º renglón\n',
                 '3º renglón\n']

    a.writelines(renglones)

    a.seek(0)      # volvemos a la posición 0

    a.readlines()   # ['1º renglón\n', ... '3º renglón\n']

except Exception as e:

    print(f'Ocurrió un error: {e}')

finally:

    a.close()      # cerramos el archivo
```



# Leyendo y escribiendo archivos

- La sentencia **with**:

```
"""
El uso de 'with' con archivos de texto constituye una buena práctica.
La ventaja es que el archivo siempre se cierra, aunque haya excepciones.
"""

with open('archivo.txt', 'r+') as a:

    data = a.read().split('\n')      # lee todo el archivo y devuelve una
                                    # lista separada por '\n'

    print(f'El archivo tiene {len(data)} renglones.')

    nueva_data = ['Lorem ipsum dolor sit amet,\n',
                  'consectetur adipiscing elit,\n',
                  'sed do eiusmod tempor incididunt\n',
                  'ut labore et dolore magna aliqua.']

    a.writelines(nueva_data)

# preguntamos si <a> fue cerrado

a.closed    # True
```

```
"""
Como open().read().split() lee el archivo entero, puede ser
perjudicial en caso de archivos demasiado grandes.

Una buena idea es recurrir a un generador para leerlo por partes.
"""

def leer_archivo(archivo):

    with open(archivo) as a:      # el modo predeterminado es 'r'

        for renglon in a:

            yield renglon

    n = 0

    for r in leer_archivo('archivo.txt'):

        n += 1

    print(f'El archivo tiene {n} renglones.)
```



# Clases

- Sintaxis y objetos de clase:

```
"""
Usamos 'class' para definir nuestras propias clases de objetos,
seguida de : sin ().

"""

class NombreDeLaClase:
    """
    Doc string.
    """

    <declaración 1>
    .
    .
    .

    <declaración n>
```

```
"""
Los objetos de clase admiten dos tipos de operaciones.

En primer lugar, referencias a sus atributos:
"""

class MiClase:
    """
    Una simple clase de ejemplo.
    """

    entero = 12345

    # Nótese el uso del argumento 'self' para hacer alusión
    # a la propia clase

    def func(self):
        """
        Devuelve un saludo.
        """
        return '¡Hola, mundo!'

objeto = MiClase()

objeto.entero    # 12345

objeto.func()    # '¡Hola, mundo!'
```



# Clases

- Instancias de clase y métodos:

```
"""
En segundo lugar, podemos crear objetos con un
estado inicial específico:
"""

class Complejo:

    def __init__(self, parte_real, parte_imag):

        self.real = parte_real
        self.imag = parte_imag

c = Complejo(3.0, 4.5)

c.real    # 3.0
c.imag    # 4.5

10 + c.real    # 13.0
```

```
"""

En el siguiente ejemplo, llamamos al método agregar_truco() y, aunque
sabemos que admite dos argumentos, sólo pasamos uno.

Python se encarga de pasarle el primero, self, que es la instancia que
lo está referenciando.
"""

class Perro:

    def __init__(self, nombre, edad):

        self.nombre = nombre
        self.edad = edad
        self.trucos = []

    def agregar_truco(self, truco):

        self.trucos.append(truco)

scooby = Perro('Scooby', 1)

scooby.trucos    # []
scooby.agregar_truco('traer el diario')
scooby.agregar_truco('morder las flores')
scooby.trucos    # ['traer el diario', 'morder las flores']
```



# Clases

- Variables de clase e instancias:

```
"""
Las variables de clase son compartidas por todos los
objetos de una misma clase.
"""

class Perro:

    reino = 'animal'
    especie = 'canina'

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

scooby = Perro('Scooby')

cometa = Perro('Cometa')

scooby.reino, scooby.especie      # ('animal', 'canina')
cometa.reino, cometa.especie      # ('animal', 'canina')

scooby.nombre      # 'Scooby'

cometa.nombre      # 'Cometa'
```

```
"""

El siguiente es un ejemplo de un uso erróneo de una variable
de clase, ya que la misma lista <trucos> es compartida entre
todas las instancias de <Perro>.

"""

class Perro:

    trucos = []

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

    def agregar_truco(self, truco):
        self.truco.append(truco)

scooby = Perro('Scooby')

cometa = Perro('Cometa')

scooby.agregar_truco('traer el diario')

cometa.agregar_truco('traer la pelota')

scooby.trucos      # ['traer el diario', 'traer la pelota']
```



# Módulos

```
"""
Ejemplo de árbol de directorios.

+ app_dir:
    - app.py
    - LICENSE
    - README.md
    - setup.py

+ comunes:
    - __init__.py
    - modulo1.py
    - modulo2.py

+ clases:
    - __init__.py
    - modulo1.py
    - modulo2.py

+ dir_con_sub_dirs:
    - __init__.py

    + sub_dir1:
        - __init__.py
        - modulo.py

    + sub_dir2:
        - __init__.py
        - modulo.py
```

```
# app.py

# los directorios se comportan como objetos con atributos (módulos)
import comunes.modulo1

# también podemos importar uno o más objetos con 'from'
from comunes.modulo2 import algo

from clases.modulo1 import Clase1A, Clase1B

# podemos definir nuevos nombres para objetos importados con 'as'
from clases.modulo2 import Clase2 as c_dos

# también podemos importar variables
from dir_con_sub_dirs.sub_dir1.modulo import var1, var2, var3

# o importar todo lo que contenga un módulo con '*'
from dir_con_sub_dirs.sub_dir_2.modulo import *

# podemos guardar funciones en variables (sin usar '()')
func1 = modulo1.func1
func2 = modulo1.func2

# y ejecutarlas más tarde
func1()
func2()
```



# Python Academy

# Fundamentals

Clase 3

¡Gracias!

