

# Python Academy

# Fundamentals

Clase 2

¡Bienvenidos!



# Temario

- Listas
  - Indexación
  - Segmentos (rebanadas)
  - Métodos
  - Listas por comprensión
- La sentencia **del**
- Funciones
  - Cómo se definen (sintaxis)
  - Argumentos
  - Expresiones *lambda*
- Excepciones



# Listas

## o Indexación:

```
# 0 1 2 3 4 5 6 7 8 --> índices positivos
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9]
# -9 -8 -7 -6 -5 -4 -3 -2 -1 --> índices negativos

"""
El índice 0 siempre nos devuelve el primer ítem de la lista.

El índice -1 siempre nos devuelve el último.

"""
print(lista[0], lista[-1]) # 1 9

"""
Podemos reasignar valores en cada índice:

"""
lista[0] = 'python'
lista[1] = 'academy'
lista[-1] = 'fundamentals'

print(lista) # ['python', 'academy', 3, 4, 5, 6, 7, 8, 'fundamentals']
```

```
lista = ['python', 'academy', 'fundamentals']

"""
list.index
"""

lista.index('python') # 0
lista.index('academy') # 1

"""
list.insert
"""

lista.insert(0, 'corro todo a la derecha')

print(lista) # ['corro todo a la derecha', 'python', ...]

"""
list.pop
"""

a = lista.pop() # borra devolviendo el valor (default: -1)
b = lista.pop(0) # acepta un índice

print(a) # fundamentals
print(b) # corro todo a la derecha
print(lista) # ['python', 'academy']
```



# Listas

- Más métodos:

```
lista = [1, 2, 2, 3, 3, 3]

"""
list.count
"""

lista.count(1)    # 1
lista.count(2)    # 2
lista.count(3)    # 3

"""

list.copy
"""

lista.copy()      # [1, 2, 2, 3, 3, 3]

"""

list.clear
"""

lista.clear()

print(lista)      # []
```

```
lista = [1, 2, 3, 4, 5, 5]

"""
list.remove
"""

lista.remove(5)

print(lista)      # [1, 2, 3, 4, 5]

"""

list.reverse
"""

lista.reverse()

print(lista)      # [5, 4, 3, 2, 1]

"""

list.sort
"""

lista.sort()

print(lista)      # [1, 2, 3, 4, 5]
```

```
lista_1 = [1, 2, 3]
lista_2 = [4, 5, 6]

"""

list.append
"""

lista_1.append(lista_2)

print(lista_1)      # [1, 2, 3, [4, 5, 6]]

"""

list.extend
"""

lista_1.extend(lista_2)

print(lista_1)      # [1, 2, 3, 4, 5, 6]
```



# Listas

- Segmentos:

```
lista = ['python', 'academy', 'fundamentals', 'listas', 'índices', 'segmentos']

"""
El último índice no se cuenta.

"""

lista[0:2]      # ['python', 'academy']

lista[4:6]      # ['índices', 'segmentos']

"""

El tercer argumento es el salto.

"""

lista[::-2]     # ['python', 'fundamentals', 'índices']

lista[1::2]      # ['academy', 'listas', 'segmentos']

"""

Si el salto es negativo, se revierte el orden.

"""

lista[2::-1]    # ['fundamentals', 'academy', 'python']

lista[::-2]      # ['segmentos', 'listas', 'academy']
```

```
cadena = 'segmento'

cadena[-len(cadena)]      # 's'
cadena[len(cadena) - 1]    # 'o'

cadena[0:8]    # 'segmento'
cadena[2:7]    # 'gment'

cadena == cadena[:]      # True
cadena is cadena[:]      # True

"""

Omitir ambos índices devuelve una copia de la lista entera.

Pero, a diferencia de las cadenas, es una COPIA y no una referencia al mismo objeto.

"""

lista = [1, 2, 3, 4, 5]
lista == lista[:]      # True
lista is lista[:]      # False
```



# Listas

- La sentencia **del**:

```
"""
Usamos 'del' para remover una posición y su respectivo valor
de una colección.
"""

lista = [1, 2, 3, 4, 5]

del lista[-1]      # borro el último ítem
del lista[0]        # borro el primer ítem

print(lista)        # [2, 3, 4]

"""

También se pueden borrar segmentos enteros.

"""

lista = ['a', 'b', 'c', 'python', 'academy', 'd', 'e', 'f']

del lista[0:3]      # borro los índices 0, 1 y 2

del lista[-1:-4:-1]  # borro los índices -1, -2 y -3

print(lista)        # ['python', 'academy']
```

```
"""
Si quisiera borrar valores específicos de una colección, podría
iterar sobre la misma, referenciando sus índices y ejecutando
'del' donde sea necesario.
"""

lista = [1, 2, 7, 3, 4, 7, 5, 6, 7]

list(range(len(lista), 0, -1))    # [9, 8, 7, 6, 5, 4, 3, 2, 1]

n = len(lista)      # 9

for i in range(n - 1, -1, -1):    # del 8 al 0

    if lista[i] == 7:

        del lista[i]

print(lista)        # [1, 2, 3, 4, 5, 6]
```



# Listas por comprensión

- Sintaxis:

```
# ejemplo 1
# calcula el cuadrado de cada número en <numeros>
# y lo mete en <cuadrados>

numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9]

cuadrados = []

for n in numeros:

    cuadrados.append(n**2)

print(cuadrados)      # [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

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

cuadrados = [n**2 for n in numeros]

print(cuadrados)      # [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
# ejemplo 2
# crea una lista con tres listas anidadas compuestas por ceros (0)

matriz = []

for n in range(3):

    lista = []

    for x in range(4):

        lista.append(0)

    matriz.append(lista)

print(matriz)      # [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

```
matriz = [[0 for x in range(4)] for n in range(3)]

print(matriz)      # [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```



# Listas por comprensión

- Condicionales: **if**

```
# ejemplo 1
# lee los ítems en <tickets> y mete en <incidentes> los que empiezan
# con 'INC'

tickets = ['INC001', 'INC002', 'CTASK001', 'RITM001', 'CHG001', 'INC003']

incidentes = []

for ticket in tickets:

    if ticket[0:3] == 'INC':

        incidentes.append(ticket)

print(incidentes)    # ['INC001', 'INC002', 'INC003']
```

```
# ejemplo 2
# genera una lista <mult_de_seis> con múltiplos de 6
# del 1 al 99

mult_de_seis = []

for n in range(1, 100):

    if n % 2 == 0:

        if n % 3 == 0:

            mult_de_seis.append(n)

print(mult_de_seis)    # [6, 12, 18, 24, ... 90, 96]
```

```
tickets = ['INC001', 'INC002', 'CTASK001', 'RITM001', 'CHG001', 'INC003']

incidents = [ticket for ticket in tickets if ticket[0:3] == 'INC']

print(incidents)    # ['INC001', 'INC002', 'INC003']
```

```
mult_de_seis = [
    n for n in range(1, 100)
    if not n % 2
    if not n % 3
]

print(mult_de_seis)    # [6, 12, 18, 24, ... 90, 96]
```



# Listas por comprensión

- Condicionales: **else**

```
# genera una lista de booleanos <par_o_impar> indicando si los
# números en <rand_nums> son pares o impares

from random import randint

rand_nums = [randint(0, 50) for n in range(10)]

par_o_impar = []

for n in rand_nums:

    if n % 2:

        par_o_impar.append(False)

    else:

        par_o_impar.append(True)

print(par_o_impar)    # [False, False, False, ... True, False]
```

```
#####
## BONUS ##
#####

"""

any() devuelve True si alguno de los valores es considerado verdadero.

"""

any(par_o_impar)      # True

any(not n % 2 for n in rand_nums)  # True

"""

all() devuelve True si todos los valores son considerados verdadero.

"""

all(par_o_impar)      # False

all(n % 2 == 0 for n in rand_nums)  # False
```

```
rand_nums = [randint(0, 50) for n in range(10)]

par_o_impar = [False if n % 2 else True for n in rand_nums]

print(par_o_impar)    # [False, False, False, ... True, False]
```



# Listas por comprensión

- Comprehensiones anidadas:

```
# ejemplo 1

listas = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

nueva = [[lista[i] for lista in listas] for i in range(3)]

print(nueva)    # [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

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

nueva = []

for i in range(3):

    temp = []

    for lista in listas:

        temp.append(lista[i])

    nueva.append(temp)

print(nueva)    # [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

```
# ejemplo 2

tickets = [['INC001', 'RITM0012', 'INC002'],
           ['INC0013', 'INC004', 'CHG0003'],
           ['INC005', 'INC006', 'CTASK004']]

incidentes = [t for lista in tickets for t in lista if 'INC' in t]

print(incidentes)    # ['INC001', 'INC002', 'INC0013',
                      #  'INC004', 'INC005', 'INC006']
```

```
tickets = [['INC001', 'RITM0012', 'INC002'],
           ['INC0013', 'INC004', 'CHG0003'],
           ['INC005', 'INC006', 'CTASK004']]

incidentes = []

for lista in tickets:

    for ticket in lista:

        if ticket.startswith('INC'):

            incidentes.append(ticket)

print(incidentes)    # ['INC001', 'INC002', 'INC0013',
                      #  'INC004', 'INC005', 'INC006']
```



# Funciones

- Sintaxis y argumentos posicionales:

```
"""
Las funciones se definen ejecutando la sentencia 'def' seguida
de un nombre* y un par de () seguido de :
```

\*Los nombres de las funciones están sujetos a las mismas reglas  
y limitaciones que los nombres de variables.

```
"""
def saludo():

    print('¡Hola!')

saludo()    # ¡Hola!
```

```
"""
Entre los () se definen las variables con las que la función va
a realizar alguna operación.
"""

def saludo(nombre):
```

```
    print(f'¡Hola, {nombre}! ')

saludo('Python Academy')    # ¡Hola, Python Academy!
```

```
"""
La sentencia 'return' se utiliza para devolver valores
al finalizar la ejecución de la función.
"""

def suma(x, y):
```

```
    """
    Devuelve la suma de x + y.
    """

    return x + y
```

```
a = suma(2, 3)

print(a)    # 5
```



# Funciones

- Argumentos predeterminados y con clave:

```
"""  
Definimos argumentos predeterminados para extender la funcionalidad  
de una función sin la necesidad de pasar todos los argumentos  
requeridos para su ejecución.  
"""  
  
def generar_saludo(nombre, imprimir=False):  
    """  
    Devuelve un saludo personalizado y lo imprime  
    si <imprimir> == True.  
    """  
    saludo = f'¡Hola, {nombre}!'  
  
    if imprimir:  
  
        print(saludo)  
  
    return saludo  
  
saludo = generar_saludo('Python Academy')      # no va a imprimir porque  
                                                # <imprimir> == False
```

```
def si_o_no(mensaje, intentos=4, error='Intentá de nuevo.'):  
    """  
    Pregunta por sí o por no hasta recibir una respuesta válida  
    o agotar el número de intentos.  
  
    Devuelve True o False, dependiendo del valor de la respuesta.  
  
    Si se pasa intentos=0, no hay límite de intentos.  
    """  
    while True:  
  
        ok = input(mensaje)  
  
        if ok.lower() in ('y', 'yes', 'n', 'no'):  
            break  
  
        intentos -= 1  
  
        if intentos == 0:  
            print('Respuesta inválida.')  
            break  
  
        print(error)  
  
    return ok.lower() in ('y', 'yes')  
  
si_o_no('¿Desea continuar? ', intentos=5)
```



# Funciones

- Listas arbitrarias de argumentos:

```
"""
<args> es una tupla que contiene todos los argumentos posicionales
que no fueron explícitamente definidos.
"""

def concatenador(*args, sep='/'):
    """
    Devuelve una cadena con los valores en <args>
    separados por <sep>.
    """
    return sep.join(args)

concatenador(' ', 'usr', 'bin', 'python')      # '/usr/bin/python'

"""
Pasamos los contenidos de una colección anteponiendo un *.
"""

comandos = ['find / -name python', 'grep -v "denied"']

concatenado = concatenador(*comandos, sep=' | ')

print(concatenado)    # find / -name python | grep -v "denied"
```

```
"""
<kwarg> es un diccionario que contiene todos los argumentos
con clave que no fueron explícitamente definidos.
"""

def args_y_kwargs(*args, **kwargs):
    """
    Imprime los tipos y contenido de <args> y <kwarg>.
    """
    print(args, type(args))
    print(kwargs, type(kwargs))

args_y_kwargs(1, 2, 3, 4, 5, user='python', pwd='academy')

# (1, 2, 3, 4, 5) <class 'tuple'>
# {'user': 'python', 'pwd': 'academy'} <class 'dict'>
```



# Funciones

## ○ Expresiones lambda:

```
"""
Podemos crear pequeñas funciones anónimas utilizando la sentencia 'lambda'.
Pueden ser empleadas en cualquier lugar donde una función sea requerida,
aunque están restringidas sintácticamente a una única expresión.
"""

# expresión lambda
suma = lambda x, y: x + y
suma(4, 3)    # 7

# función convencional
def suma(x, y):
    return x + y

suma(4, 3)    # 7
```

```
# ejemplos

def incrementador(n):
    """
    Devuelve una función que incremente
    <x> según lo que valga <n>.
    """
    return lambda x: x + n

inc = incrementador(100)

inc(10)      # 110
inc(100)     # 200

"""
La función filter() se utiliza para filtrar una
colección según un determinado criterio.
"""

lista = [1, 2, 3, 4, 5]
pares = filter(lambda x: x % 2 == 0, lista)
list(pares)   # [2, 4]

"""
La función map() se utiliza para reescribir los ítems
de una colección según un determinado criterio.
"""

cuadrados = map(lambda x: x**2, lista)
list(cuadrados)  # [1, 4, 9, 16, 25]
```



# Excepciones

- Atajando excepciones: **try** y **except**

```
"""
Python nos permite definir acciones en caso de encontrar
excepciones durante la ejecución del código.
"""

while True:

    # usamos 'try' para lo que queremos intentar

    try:

        entero = int(input('Ingresá un número: '))

        break

    # y 'except' para atajar excepciones

    except ValueError:

        print('Valor no válido. Intentá de nuevo...')
```

```
"""
'except' también admite múltiples tipos de excepciones:
"""

try:
    # acá va algo
    pass
except (TypeError, NameError, ValueError):
    pass
```



# Excepciones

- Atajando excepciones: **except** múltiples y **else**

```
"""
Definimos distintas acciones según el tipo de excepción:
"""

def division(a, b): return a / b

while True:

    try:

        num = float(input('\nIngresá un número divisor: '))

        print(f'El resultado es: {division(8, num)}')

        break

    except ValueError:

        print('El valor ingresado no es un número.')

    except ZeroDivisionError:

        print('No se puede dividir por cero.')

    except Exception as e:

        print(f'Error no previsto: {e}'')
```

```
"""
Usamos 'else' para definir acciones en caso de que
no hayan ocurrido excepciones.
"""

def division(a, b):

    try:

        resultado = a / b

    except ZeroDivisionError:

        print('No se puede dividir por cero.')

    else:

        print(f'El resultado es: {resultado}')

division(4, 2)      # El resultado es: 2.0
division(4, 0)      # No se puede dividir por cero.
```



# Excepciones

- Definiendo acciones de limpieza: **finally**

```
"""
Usamos 'finally' para ejecutar acciones al final del bloque try ... except, haya excepciones o no.
"""

while True:

    try:

        # acá se hace algo

        pass

    except:

        print('Algo salió mal.')

    else:

        print('Nada salió mal.')

    finally:

        print('Fin del bloque try ... except.')  
'''
```



# Excepciones

- Levantando excepciones: **raise**

```
"""
Usamos 'raise' para levantar excepciones.
"""

def func(*args):

    accepted = {'arg1', 'arg2', 'arg3'}

    if any(type(arg) != str for arg in args):      # si algún argumento no es str
        raise TypeError('Los argumentos deben ser del tipo str.')

    if any(arg not in accepted for arg in args):      # si algún argumento no está
                                                       # en <accepted>

        raise ValueError('Alguno de los argumentos provistos es inválido.')

    return True

try:

    func('arg4')      # pasamos un argumento no válido

except Exception as e:

    print(f'Hubo un error: {e}'')
```

```
"""
También podemos definir nuestras propias excepciones.

Todas las excepciones provienen de la misma base: Exception.
"""

class CustomExceptionA(Exception):
    pass

class CustomExceptionB(Exception):
    pass

for cls in [CustomExceptionA, CustomExceptionB]:

    try:

        raise cls()

    except CustomExceptionA:

        print('A')

    except CustomExceptionB:

        print('B')
```



# Excepciones

- Afirmando condiciones: **assert**

```
"""
Las afirmaciones son sentencias que, valga la redundancia,
afirman algún hecho confidencialmente.

Si la condición se cumple, el programa sigue.

Si no, levanta una excepción del tipo AssertionError.
"""

def division(dividendo, divisor):

    assert divisor != 0

    return dividendo / divisor

a = 90
b = 0

try:

    division(a, b)

except Exception as e:

    print(type(e).__name__)    # AssertionError
```

```
"""
Para agregar un mensaje a la excepción, podemos pasarlo directamente
después de la condición, separado por una , (coma).
"""

def division(dividendo, divisor):

    assert divisor != 0, 'El divisor no puede ser cero.'

    return dividendo / divisor

a = 90
b = 0

try:

    division(a, b)

except Exception as e:

    print(e)    # El divisor no puede ser cero.
```



# Python Academy

# Fundamentals

Clase 2

¡Gracias!

