UNIVERSIDAD TECNOLOGICA NACIONAL

Capítulo 4: Funciones

Programación 2

Funciones

```
def dividir(x, y):     # Función y sus parámetros
    return x / y


def sin_return(x, y):  # Por defecto se devuelve None
    x / y


def dividir(x: float, y: float) -> float:     # Type-Hints |
                                                RECOMENDADO
    return x / y


def sin_return(x: float, y: float) -> float:  # Recomendaciones
                                                con Type-Hints
    x / y
```

Funciones

```
def dividir(x, y):     # Función y sus parámetros
    return x / y


dividir(10, 8) == 1.25      # Orden idéntico a la
                                definición

dividir(8, 10) == 0.8       # El Orden es importante

dividir(x=10, y=8) == 1.25  # Usando parámetros
                                explícitos

dividir(y=8, x=10) == 1.25  # Orden irrelevante
```

Funciones

```
def es_mayor_de_edad(edad: int, limite: int = 18) -> bool:  # Valor por defecto
    if edad >= limite:
        resultado = True
    else:
        resultado = False
    return resultado


def es_mayor_de_edad(edad: int, limite: int = 18) -> bool:  # Múltiples returns
    if edad >= limite:
        return True
    return False


def es_mayor_de_edad(edad: int, limite: int = 18) -> bool:  # Return expression
    return edad >= límite
```

Funciones

```
from typing import List, Tuple   # Biblioteca Estándar

precios: List[float] = [4.04, 5.37, 7.77, 0.09, 9.11, 4.96, 9.12,
2.28, 8.09, 7.36]


# Return con múltiples valores

def hay_oferta(precios: List[float]) -> Tuple[bool, float]:

    precio_mas_bajo = min(precios)

    if precio_mas_bajo < 3:

        return True, precio_mas_bajo

    return False, precio_mas_bajo


hay_oferta(precios) == (True, 0.09)  # => Devuelve Tupla

existe_oferta, monto = hay_oferta(precios)
                                                # =>Desempaquetado
```

Parámetros Arbitrarios

```
def suma(*args: float):          # Parámetros posicionales arbitrarios
    resultado = 0
    for valor in args:
        resultado += valor
    return resultado


suma(1, 2, 3) == 6
```

```
def concatenate(**kwargs: str):   # Parámetros de palabra clave
                                    arbitrarios
    return " ".join(kwargs.values())


concatenate(a="Hola", b="Mundo")  # => 'Hola Mundo '
```

Funciones de orden superior

```
from typing import Callable   # Biblioteca Estándar

def aplicar_funcion(lista: List[float], funcion: Callable[[float], float]) -> List[float]:

(el tipo de dato especificado dentro del callable es: el primero es el tipo de dato del parámetro que se pasa y el segundo el tipo de dato de lo que retorna esta función)

    resultados = []

    for elemento in lista:

        resultado = funcion(elemento)

        resultados.append(resultado)

    return resultados



def cuadrado(x: float) -> float:

    return x ** 2



lista: List[float] = [1, 2, 3, 4, 5, 6]

aplicar_funcion(lista, cuadrado)
```

Closures: funciones dentro de funciones

```
def elevar(y: float) -> Callable[float, float]:

    def auxiliar(x: float) -> float:
        return x ** y

    return auxiliar
```

```
número: float =  2
elevar_cuadrado: Callable[float, float] = elevar(2)
aplicar_funcion(numero, elevar_cuadrado) ==  4
```

Evaluación parcial

```
from functools import partial  # Biblioteca estandar


def elevar_xy(x: float, y: float) -> float:
    return x ** y



numero: float = 2

elevar_cuadrado_parcial: Callable[float, float] =
partial(elevar_xy, y=2)

aplicar_funcion(numero, elevar_cuadrado_parcial) == 4
```

Lambdas: funciones anónimas

```
numero: float = 4

aplicar_funcion(numero, lambda x: x**2) == 4
```

Funciones sobre funciones: map, filter y reduce

```
from functools import reduce    # Biblioteca estándar

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

cuadrados = map(lambda x: x ** 2, lista)    # => [1, 4, 9, 16, 25, 36]


cuadrados_pares = filter(lambda x: x > 5, cuadrados)
                                                    # => [9, 16, 25, 36]


suma_pares = reduce(lambda x, y: x + y, cuadrados_pares)
                                                    # => 86
```

Caso de uso: tenemos una lista de números, queremos elevarlos al cuadrado. No nos interesan aquellos números que luego de elevarlos al cuadrado sean mayores a 5 y luego queremos obtener la suma.

Comprensiones

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


cuadrados_: = [elevar_cuadrado(x) for x in lista]
                                        # => [1, 4, 9, 16, 25, 36]


cuadrados_pares_ = [x for x in cuadrados_ if x > 5]
                                            # => [9, 16, 25, 36]


suma_pares = sum(cuadrados_pares_)              # => 86
```

Ejemplo llevado a una sola expresión.

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

suma_pares: float = sum(elevar_cuadrado(x) for x in lista if
elevar_cuadrado(x) > 5)
```