

Paradigma Funcional

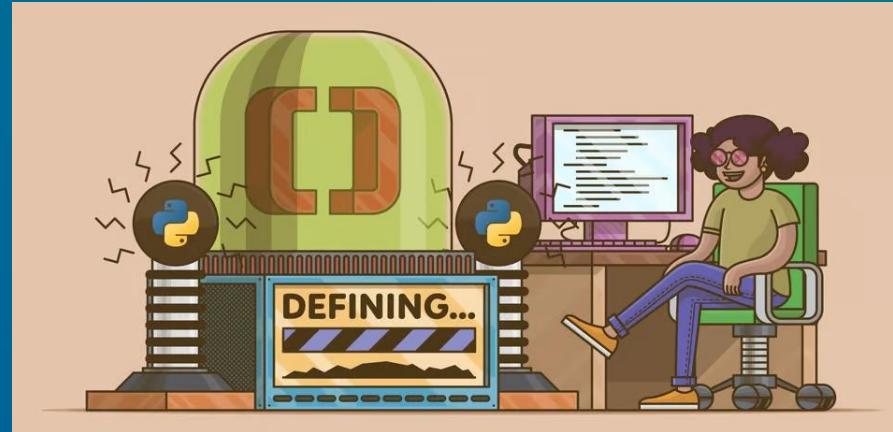


- El concepto **programación funcional** hace referencia a un estilo de programación que utiliza las funciones como unidad básica de código.
- Hay desde lenguajes puramente funcionales como Haskell o Lisp, hasta lenguajes multiparadigma como **Python**.
- Para que un lenguaje permita la programación funcional, **debe tratar a las funciones como ciudadanos de primera clase**. Es lo que ocurre con Python; las funciones son objetos, al igual que los strings, los números y las listas. Las funciones pueden pasarse como parámetros a otras funciones o devolverse como valores de retorno de funciones.

Características principales:

- Funciones puras: Una función pura es aquella cuyo resultado depende solo de sus parámetros de entrada y no tiene efectos secundarios.
- Hacer código más declarativo: Se enfoca en el "qué" en lugar del "cómo", lo que suele resultar en un código más claro y fácil de entender.
- Funciones de orden superior: Estas son funciones que pueden recibir otras funciones como argumentos o devolver funciones como resultado.
- Expresiones en lugar de instrucciones: En lugar de usar bucles y comandos, se favorecen las expresiones y combinaciones de funciones.

Funciones como Ciudadanos de Primera Clase



Se dice que un lenguaje de programación tiene funciones de primera clase si se admite:

- Asignar funciones a variables o almacenarlas en estructuras de datos.
- Poder pasar funciones como argumentos a otras funciones.
- Que el valor de retorno de una función sea otra función.

Las funciones en Python son ciudadanos de primera clase (first-class citizens). La máxima de Python de “**Everything is an object**” (todo es un objeto) también vale para las funciones.

- Asignar funciones a variables

```
● ● ●  
  
def sumar(a, b):  
    return a + b  
  
operacion = sumar  
  
print(operacion(4, 2)) # 6
```

Paradigma funcional

- Almacenarlas en estructuras de datos (listas)

```
● ● ●

def sumar(a, b):
    return a + b

def restar(a, b):
    return a - b

def multiplicar(a, b):
    return a * b

operaciones = [sumar, restar, multiplicar]

for operacion in operaciones:
    print(operacion(10, 5))
```

- Almacenarlas en estructuras de datos (tuplas)

```
● ● ●

def sumar(a, b):
    return a + b

def restar(a, b):
    return a - b

def multiplicar(a, b):
    return a * b

operaciones = (sumar, restar, multiplicar)

resultado_suma = operaciones[0](10, 5)
resultado_resta = operaciones[1](10, 5)

print(f"Resultado de la suma: {resultado_suma}")
print(f"Resultado de la resta: {resultado_resta}")
```

- Almacenarlas en estructuras de datos (diccionarios)

```
● ● ●

def sumar(a, b):
    return a + b

def restar(a, b):
    return a - b

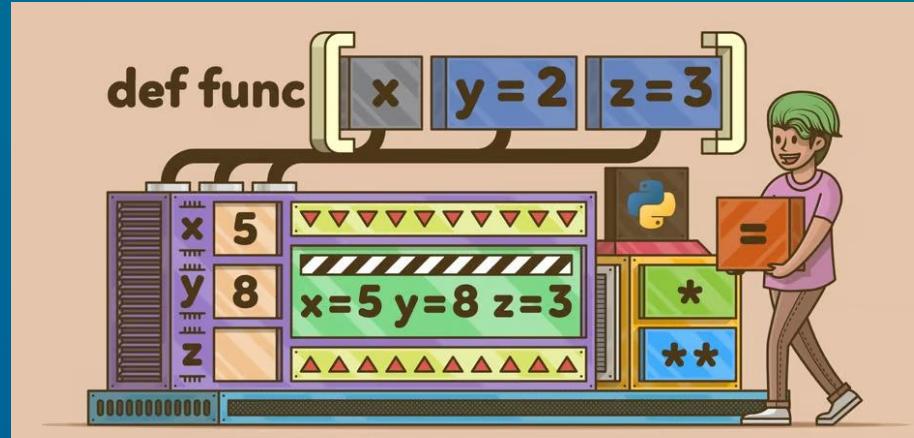
def multiplicar(a, b):
    return a * b

operaciones = {
    "sumar": sumar,
    "restar": restar,
    "multiplicar": multiplicar
}

resultado = operaciones["multiplicar"](10, 5)
print(f"Resultado de la multiplicación: {resultado}")
```

Esto demuestra la flexibilidad de las funciones como ciudadanos de primera clase en Python, permitiendo manipularlas y almacenarlas como cualquier otro tipo de dato.

Funciones de Orden Superior



Paradigma funcional

Las funciones de orden superior (o funciones de mayor orden) son aquellas que pueden tomar otras funciones como argumentos y/o devolver funciones como resultados. Son una característica poderosa que permite una gran flexibilidad y funcionalidad en el código.

Funciones como parámetro de otras funciones

En Python, las funciones pueden ser utilizadas como parámetros de otras funciones. Esto es posible porque las funciones en Python, como mencionamos anteriormente, son ciudadanos de primera clase.

- Función que toma otra función como parámetro



```
def sumar(x, y):
    return x + y

def restar(x, y):
    return x - y

def operar(a, b, funcion):
    return funcion(a, b)

resultado_suma = operar(10, 5, sumar)
resultado_resta = operar(10, 5, restar)

print(f"Resultado de la suma: {resultado_suma}") # Resultado de la suma: 15
print(f"Resultado de la resta: {resultado_resta}") # Resultado de la resta: 5
```



```
def aplicar_operacion(numero, funcion):
    resultado = funcion(numero)
    return resultado

def duplicar(x):
    return x * 2

def triplicar(x):
    return x * 3

# Usando la función aplicar_operacion
resultado1 = aplicar_operacion(5, duplicar)
resultado2 = aplicar_operacion(5, triplicar)

print(resultado1) # Imprime 10
print(resultado2) # Imprime 15
```

- Función que toma otra función como parámetro

```
● ● ●

def ordenar_lista(funcion, lista):
    """Ordena una lista según una función de comparación."""
    lista.sort(key=funcion)
    return lista

def ordenar_por_longitud(cadena):
    return len(cadena)

palabras = ["manzana", "banana", "pera"]
resultado = ordenar_lista(ordenar_por_longitud, palabras)
print(resultado)
# Salida: ['pera', 'banana', 'manzana']
```

- Función que toma otra función como parámetro

```
def filtrar_lista(funcion, lista):
    """Filtrá una lista según una función de predicado."""
    resultado = []
    for elemento in lista:
        if funcion(elemento):
            resultado.append(elemento)
    return resultado

def es_par(numero):
    return numero % 2 == 0

numeros = [1, 2, 3, 4, 5]
resultado = filtrar_lista(es_par, numeros)
print(resultado)
# Salida: [2, 4]
```

Funciones como retorno de otras funciones

Es posible también, que una función retorne otra función como resultado. Este tipo de funciones se llaman funciones que retornan funciones. Este tipo de funciones son muy útiles para crear decoradores o para generar funciones personalizadas.

- Función que devuelve otra función

```
● ● ●  
  
def funcion_externa(x):  
    return x + 5  
  
def retornar_funcion():  
    return funcion_externa  
  
# Obtener la función externa  
mi_funcion = retornar_funcion()  
  
# Usar la función externa  
resultado = mi_funcion(10)  
print(resultado) # Salida: 15
```

```
● ● ●  
  
def crear_multiplicador_con_base(base):  
    def multiplicar_con_base(x):  
        return base * x  
    return multiplicar_con_base  
  
# Crear una función que multiplica por 10  
multiplicador_por_10 =  
crear_multiplicador_con_base(10)  
  
# Usar la función generada  
resultado = multiplicador_por_10(7)  
print(resultado) # Salida: 70
```

- Función que devuelve otra función que multiplica su entrada por una base dada.

```
● ● ●

def crear_multiplicador_con_base(base):
    def multiplicar_con_base(x):
        return base * x
    return multiplicar_con_base

# Crear una función que multiplica por 10
multiplicador_por_10 =
crear_multiplicador_con_base(10)

# Usar la función generada
resultado = multiplicador_por_10(7)
print(resultado) # Salida: 70
```

- Función que retorna diferentes funciones de formateo de cadenas

1

```
def formatear_mayusculas(texto):
    return texto.upper()

def formatear_minusculas(texto):
    return texto.lower()

def formatear_titulo(texto):
    return texto.title()

# Función que retorna la función de formateo deseada
def obtener_formateador(tipo):
    if tipo == 'mayusculas':
        return formatear_mayusculas
    elif tipo == 'minusculas':
        return formatear_minusculas
    elif tipo == 'titulo':
        return formatear_titulo
    else:
        raise ValueError("Tipo de formateo no válido")
```

2

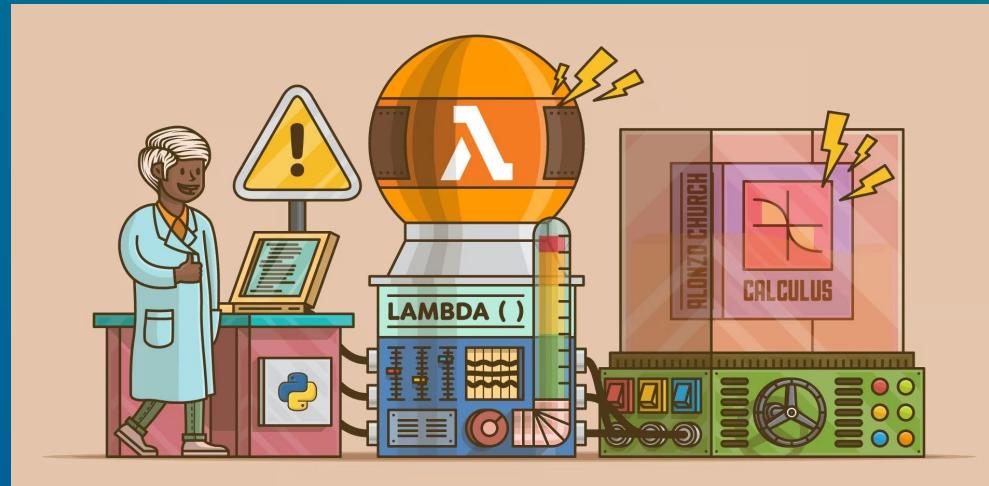
```
# Ejemplo de uso
texto = "hola mundo"

# Obtener la función de formateo deseada
formateador = obtener_formateador('mayusculas')
resultado = formateador(texto)
print(f"Texto en mayúsculas: {resultado}") # Salida: HOLA MUNDO

formateador = obtener_formateador('minusculas')
resultado = formateador(texto)
print(f"Texto en minúsculas: {resultado}") # Salida: hola mundo

formateador = obtener_formateador('titulo')
resultado = formateador(texto)
print(f"Texto en título: {resultado}") # Salida: Hola Mundo
```

Funciones lambda



Las funciones lambda o anónimas son un tipo de funciones en Python que típicamente se definen en una línea y cuyo código a ejecutar suele ser pequeño.

Características

- Funciones anónimas: Las funciones lambda son funciones anónimas, es decir, no tienen un nombre definido.
- Sintaxis concisa: Se definen en una sola línea utilizando la palabra clave lambda seguida de los parámetros, un signo : y una expresión.
- Retorno implícito: devuelven automáticamente el resultado de la expresión, sin necesidad de utilizar la palabra clave return.
- Limitación en la lógica: Están diseñadas para operaciones simples y no soportan múltiples líneas o declaraciones complejas.

Características

- No recomendadas para lógica compleja: Debido a su sintaxis limitada, no se recomienda usarlas para lógica compleja o extensa.
- Útiles en expresiones cortas: Son ideales para ser usadas en casos donde se requiere una función pequeña y de uso inmediato.
- Usos comunes: Se utilizan frecuentemente en funciones de orden superior como map(), filter(), y sorted(), donde se requieren funciones cortas.

- Sintaxis de una función lambda



```
suma = lambda a, b: a + b  
  
print(suma(2, 4)) # 6
```



```
suma = (lambda a, b, c: a + b + c)(2, 3, 4)  
  
print(suma) # 9
```

Si una función lambda no se guarda en una variable o no se pasa directamente como argumento a otra función, se crea pero no se puede utilizar porque no está referenciada en ninguna parte.

- Comparando una función normal con una función lambda

```
● ● ●  
# función normal  
def suma(a, b):  
    return a + b  
  
resultado = suma(3, 5)  
print(resultado) # Salida: 8
```

```
# función lambda  
suma_lambda = lambda a, b: a + b  
  
resultado_lambda = suma_lambda(3, 5)  
print(resultado_lambda) # Salida: 8
```

- Lo que sería una función que suma dos números como la que vemos primero en el código.
- Se podría expresar en forma de una función lambda.

- lambda como parámetro de una función



```
def validar(valor, condicion):
    return condicion(valor)

es_positivo = lambda x: x > 0
print(validar(10, es_positivo)) # Salida: True
print(validar(-5, es_positivo)) # Salida: False
```

En este ejemplo la función validar acepta dos parámetros, uno es un entero y el otro una función. En este caso le pasamos una función lambda como segundo parámetro.

Resumen de la clase:

- Funciones como ciudadanos de primera clase: Las funciones en Python pueden ser asignadas a variables, pasadas como argumentos y retornadas como resultado.
- Funciones como parámetros: Permiten crear soluciones más flexibles y reutilizables al pasar funciones como argumentos.
- Funciones con retorno de otras funciones: Permiten generar funciones especializadas en tiempo de ejecución.
- Funciones lambda: Son expresiones concisas para crear funciones anónimas, útiles en situaciones donde se requieren funciones simples y cortas.

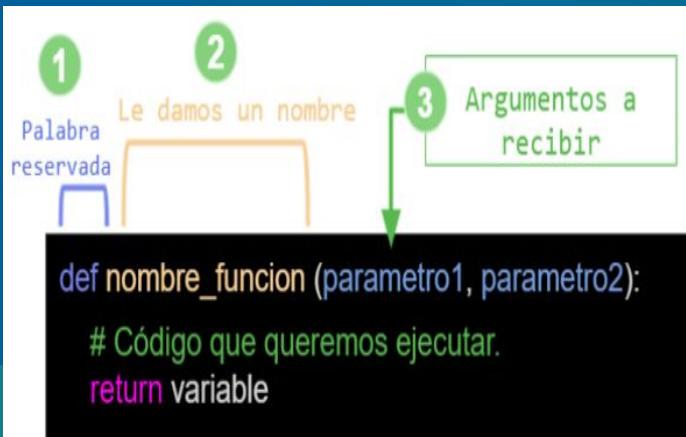
Recomendaciones:

- **Practicar** el uso de funciones como ciudadanos de primera clase.
- **Pasar** funciones como parámetros para mayor flexibilidad.
- **Explorar** el retorno de funciones para crear lógica modular.
- **Dominar** las funciones lambda para mantener el código limpio.

Definición de función

Una función es un fragmento de código o bloque de código que realiza una serie de tareas que devuelven (return) o no (sin return) un resultado. Las funciones nos ayudan a dividir el programa en partes, lo que facilita el diseño, la depuración, la reutilización, la escalabilidad debido a que podemos de manera puntual realizar alguna modificación si así lo requiriera nuestro código.

Las funciones para alcanzar su objetivo pueden recibir datos (entrada) y devolver datos (salida). Comúnmente llamamos a estas entradas **parámetros** o **argumentos**, vemos un ejemplo.



```
funcionesPy.py > ...
# Cuando creamos una funcion asignamos
# que tipo de PARAMETROS recibira
Codeium: Refactor | Explain | Generate Docstring | ×
def datos_usuario(nombre, apellido, edad):
    return "Tu nombre es:", nombre,
           "Tu apellido es:", apellido,
           "Tu edad es:", edad

# Invocamos a la funcion pasamos
# los ARGUMENTOS que le vamos a dar

print(datos_usuario("Fernando", "Valls", 33))
```

Tu nombre es: Fernando
Tu apellido es: Valls
Tu edad es: 33

Ejemplo de funciones nativas

En python contamos con funciones nativas como por ejemplo:

- print("cadena de texto") -> mostramos por pantalla un mensaje
- len() -> podemos conocer el tamaño de un texto
- input() -> lo utilizamos para ingresar por teclado datos.

```
print("Hola mundo!")
texto = "esto es un texto"
print("Tamaño del texto: ",len(texto))
nombre = input("Ingresa tu nombre: ")
print("Tu nombre es: ",nombre)
```

```
Hola mundo!
Tamaño del texto: 16
Ingresa tu nombre: Fernando
Tu nombre es: Fernando
```

Funciones con retorno

Como hemos visto todas las funciones en python tienen un “return”, a continuación adjuntamos la lista que provee la documentación oficial “[Doc python](#)”

Built-in Functions

A	E	L	R
abs()	enumerate()	len()	range()
aiter()	eval()	list()	repr()
all()	exec()	locals()	reversed()
anext()			round()
any()			
ascii()			
B	F	M	S
bin()	filter()	map()	set()
bool()	float()	max()	setattr()
breakpoint()	format()	memoryview()	slice()
bytearray()	frozenset()	min()	sorted()
bytes()			
C	G	N	O
callable()	getattr()	next()	staticmethod()
chr()	globals()		str()
classmethod()			sum()
compile()	hasattr()	object()	super()
complex()	hash()	oct()	
D	H	open()	T
delattr()	help()	ord()	tuple()
dict()	hex()		type()
dir()	I	P	V
divmod()	id()	pow()	vars()
	input()	print()	
	int()	property()	Z
	isinstance()		zip()
	issubclass()		
	iter()		__import__()

Utilización de funciones en variable

Podemos utilizar el valor de retorno de la función para asignarlo a una variable para luego reutilizarla en nuestro código

```
def suma (valorA, valorB):
    return valorA + valorB

# Guardamos el valor en la variable
# "resultado"
resultado = suma(4, 8)
# Mostramos el valor
print("El resultado de la suma es: ", resultado)
```

Funciones Lambda

Una función lambda es una función anónima, es decir, una función sin un nombre explícito, que se puede definir en una sola línea de código. Se utilizan para realizar tareas pequeñas y específicas, y son especialmente útiles cuando se necesita pasar una función como argumento a otra función, como en el caso de las funciones de orden superior.

Cómo se crean las lambdas

En Python, las funciones lambda se crean utilizando la palabra clave `lambda`, seguida de los argumentos, un `:` y luego la expresión que se evaluará y retornará. A continuación te explico los elementos clave:

- `lambda`: Palabra clave para definir una función lambda.
- Argumentos: Una lista separada por comas de los argumentos que la lambda tomará.
- Expresión: Una expresión que será evaluada y retornada por la lambda.

```
# Creamos una lambda que suma 2 numeros
suma = lambda x, y: x + y
```

```
# Uso de la lambda
resultado = suma(5, 3)
# Mostramos el resultado
print("El resultado es:", resultado)
```

```
# Creamos una lambda que suma 2 numeros
pasar_a_mayuscula = lambda palabra: palabra.upper()
```

```
# Uso de la lambda
resultado = pasar_a_mayuscula("Hola mundo")
# Mostramos el resultado
print("El resultado es:", resultado)
```

Uso de funciones de orden superior

Las funciones lambda son muy útiles cuando se combinan con funciones como map, filter, y sorted, que son funciones de orden superior (funciones que aceptan otras funciones como argumentos).

Cómo se crean las lambdas

En Python, las funciones lambda se crean utilizando la palabra clave lambda, seguida de los argumentos, un : y luego la expresión que se evaluará y retornará. A continuación te explico los elementos clave:

- lambda: Palabra clave para definir una función lambda.
- Argumentos: Una lista separada por comas de los argumentos que la lambda tomará.
- Expresión: Una expresión que será evaluada y retornada por la lambda.

```
# Lista de números
números = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# Ejemplo usando map
nueva_lista_numeros_potenciados = list(map(lambda x: x ** 2, números))
print(nueva_lista_numeros_potenciados)
# Salida: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

# Ejemplo con filter
nueva_lista_solo_pares = list(filter(lambda x: x % 2 == 0, números))
print(nueva_lista_solo_pares)
# Salida: [2, 4, 6, 8, 10]

estudiantes = [
    {"nombre": "Ana", "edad": 23},
    {"nombre": "Juan", "edad": 21},
    {"nombre": "María", "edad": 25},
    {"nombre": "Pedro", "edad": 19},
    {"nombre": "Luis", "edad": 27},
    {"nombre": "Carlos", "edad": 22},
    {"nombre": "Laura", "edad": 24},
    {"nombre": "Marta", "edad": 20},
    {"nombre": "Jorge", "edad": 26},
    {"nombre": "Sandra", "edad": 18},
]

# Ordenar por edad
estudiantes_ordenados = sorted(estudiantes, key=lambda x: x["edad"])
print(estudiantes_ordenados)

# Ordenar por nombre
estudiantes_ordenados = sorted(estudiantes, key=lambda x: x["nombre"])
print(estudiantes_ordenados)
```

Variables locales vs. globales

Variables locales: Son aquellas que se definen dentro de una función y solo son accesibles en el ámbito de esa función. Estas variables "nacen" cuando la función se invoca y "mueren" cuando la función termina su ejecución. No se pueden acceder ni modificar fuera de la función donde fueron definidas.

Variables globales: Son aquellas que se definen fuera de todas las funciones y son accesibles desde cualquier parte del programa. Sin embargo, para modificar una variable global dentro de una función, es necesario declarar explícitamente como global dentro del ámbito de esa función utilizando la palabra clave global.

Scope: El scope (o ámbito) se refiere al contexto en el que una variable es visible y accesible. En el caso de las variables locales, su scope está limitado a la función en la que se definen. Las variables globales, en cambio, tienen un scope más amplio, ya que pueden ser accesibles desde cualquier parte del programa.

```
# Definimos x como variable global
x = 10
Codeium: Refactor | Explain | Generate Docstring | ×
def funcion_local():
    # Definimos x como variable local
    x = 20
    print("Soy una variable local:", x)

# Imprimimos la variable global
print("Soy una variable global:", x)
# Imprimimos la variable local
funcion_local()
# Volvemos a imprimir la variable global
print("Sigo siendo una variable global:", x)
```

```
Soy una variable global: 10
Soy una variable local: 20
Sigo siendo una variable global: 10
```