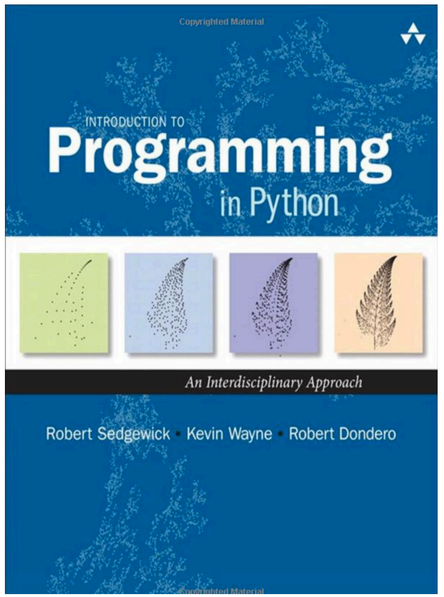
Taller de Programación

Clase 10: Funciones

Daniela Opitz, Diego Caro
dopitz@udd.cl



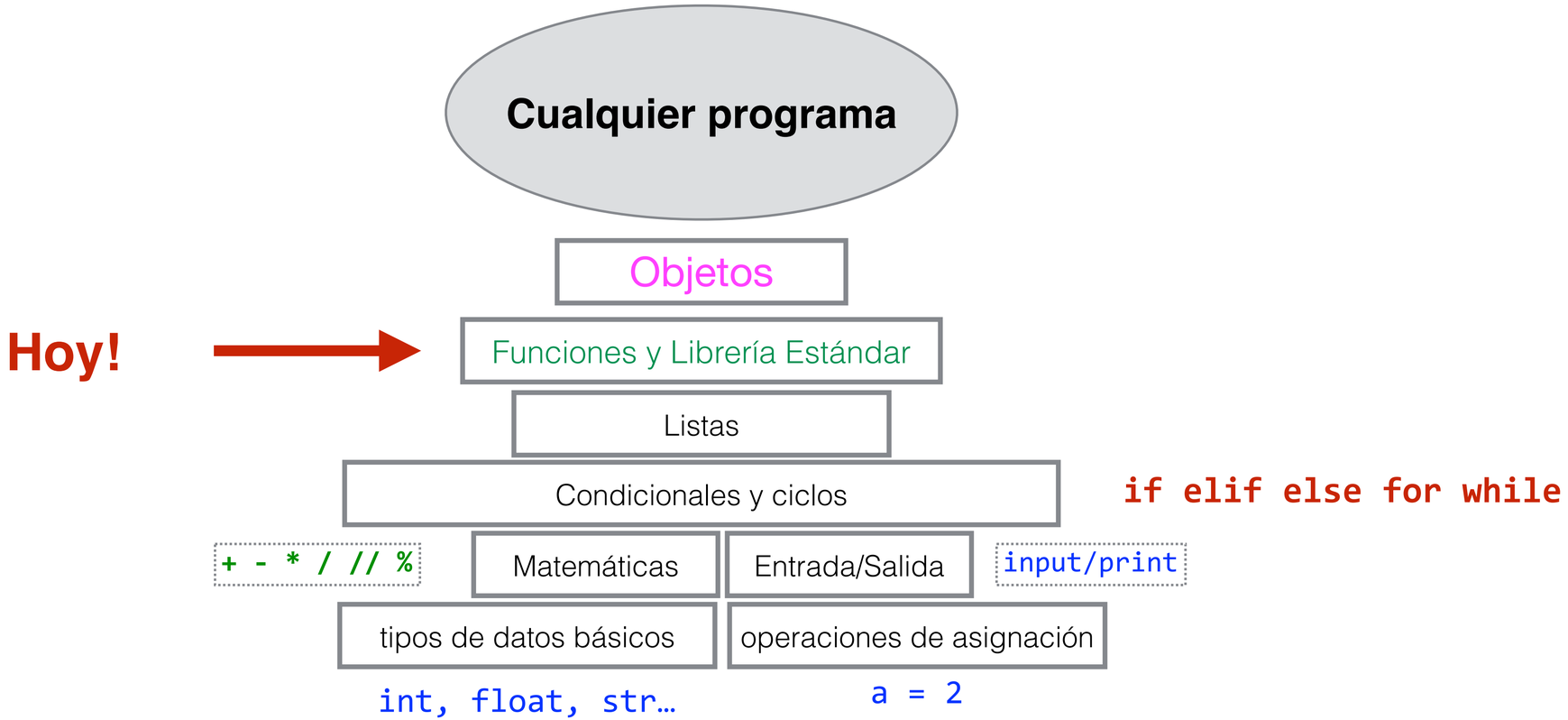
Basada en presentaciones oficiales de libro Introduction to Programming in Python (Sedgewick, Wayne, Dondero).

Disponible en https://introcs.cs.princeton.edu/python

Outline

- Funciones
- Funciones que retornan datos
- Funciones que no retornan datos
- Funciones en la librería estándar: sum, min, max, etc
- Scope: variables locales, variables globales

¿Dónde estamos?



Funciones

En computación una **función** es un subprograma o subrutina que realiza una tarea específica y a veces devuelve un valor.

- Funciones que retornan datos

- Funciones que no retornan datos pero realizan algún procedimiento

Las funciones se vuelven útiles cuando empezamos a definir las nuestras, organizando pedazos de código por funcionalidad y rehusándolo en múltiples lugares del programa principal. En Python, las funciones se definen con la instrucción **`def`**.

Funciones que retornan datos

Por ejemplo, podemos encapsular una función que calcule el promedio de la siguiente manera:

Función

Bloque de código que se ejecuta cuando se solicita.

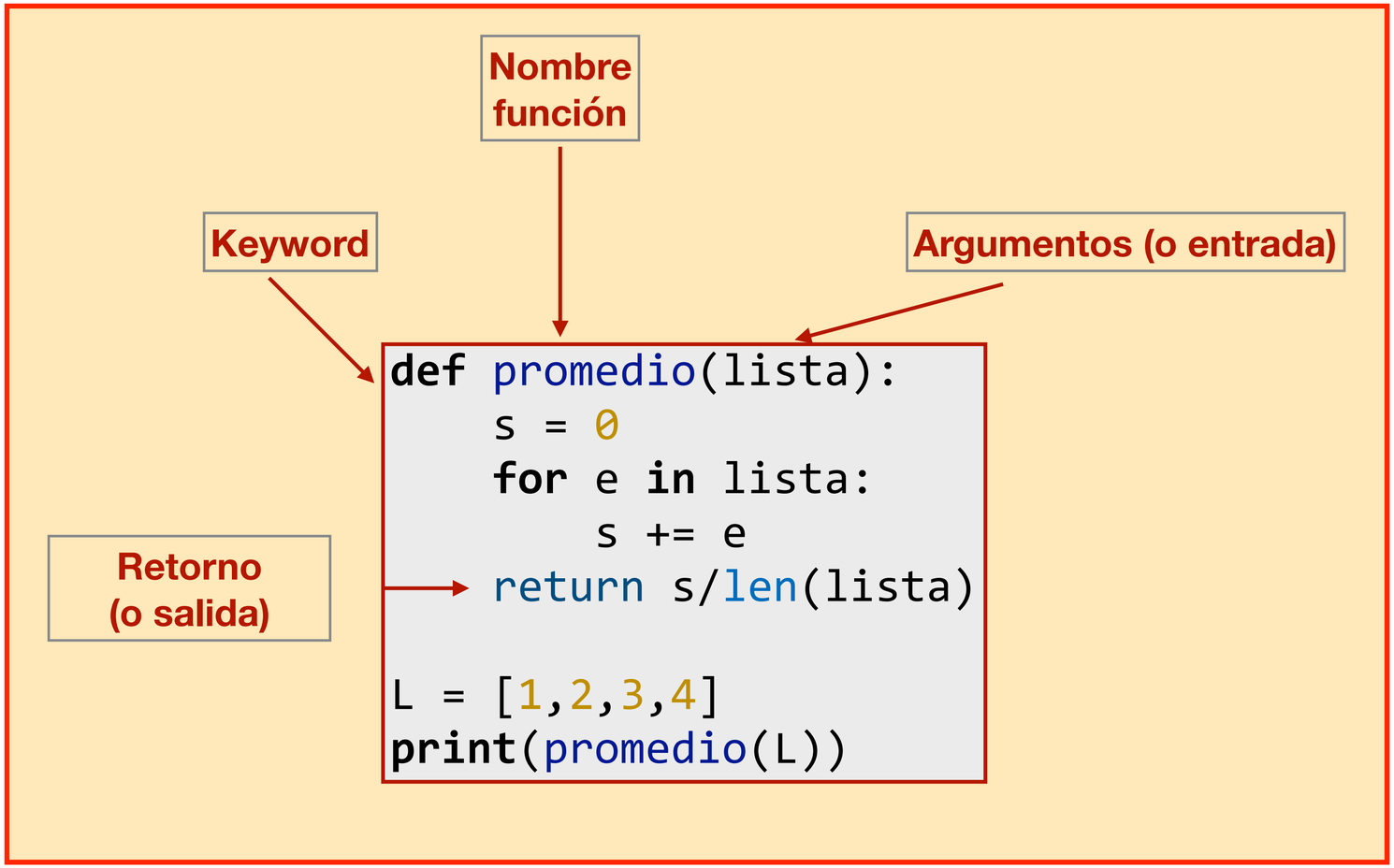
```
def promedio(lista):
    s = 0
    for e in lista:
        s += e
    return s/len(lista)

L = [1,2,3,4]
print(promedio(L))
```

Objetivo

Dividir el programa en partes más pequeñas que se pueden reutilizar.

Funciones que retornan datos



Funciones que retornan datos

return

- Se utiliza para devolver el resultado de ejecutar el algoritmo de la función.
- La función se detiene (incluso si return está en un ciclo!).
- Si una función no tiene return, por defecto devuelve `None`.

```
def promedio(lista):
    s = 0
    for e in lista:
        s += e
    return s/len(lista)

L = [1,2,3,4]
print(promedio(L))
```

Ver demo en https://goo.gl/VXUBGC

Ejemplo. Números primos

```
1 def es_primo(n):
2     if n < 2: return False
3     i = 2
4     while i*i <= n:
5         if n % i == 0: return False
6         i += 1
7     return True
8
9 print('9 es primo?', es_primo(9))
```

Ejemplo. Fibonacci

```
def fibonacci(N):
    L = []
    a=0
    b=1
    while len(L) < N:
        c = a
        a = b
        b = c + b
        L.append(a)
    return L
```

Multiples argumentos

- Las funciones pueden recibir múltiples argumentos

```
def suma(a, b):
    c = a+b
    return c

s = suma(2, 5)
print(s)
7
```

Actividad

Calcule la variación en el tiempo de viaje

Día	Tiempo de viaje en minutos
1	67
2	45
3	84
s	**19,553**

Día	Tiempo de viaje en minutos
1	70
2	70
3	70
s	**0**

Desviación Estándar

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

Promedio

Usando funciones

```
 1 from math import sqrt
 2
 3 L = [67, 45, 84]
 4
 5 s = 0
 6 for e in L:
 7     s += e
 8 prom = s / len(L)
 9
10 d = 0
11 for e in L:
12     d += (e - prom)*(e - prom)
13 sd = sqrt(d / (len(L)-1))
14 print(sd)
```

```
 1 from math import sqrt
 2
 3 def prom(seq):
 4     return sum(seq)/len(seq)
 5
 6 def desviacion(seq):
 7     p = prom(seq)
 8     d = 0
 9     for e in seq:
10         d += (e - p)*(e - p)
11     return sqrt(d / (len(seq)-1))
12
13 L = [67, 45, 84]
14 print(desviacion(L))
```

Funciones que no retornan datos

Por ejemplo, **`print`** es una función (en Python 3, al menos) que realiza la acción de imprimir pero no retorna un dato.

```
print('abc')
```

Aquí, **`print()`** es el nombre de la función, y `'abc'` es lo que se llama un **argumento** de la función.

Podríamos definir la función suma así:

```
def suma(a, b):
    c = a+b
    print(c)

suma(2, 5)
7
```

Nos sirve esta función?

Argumentos clave

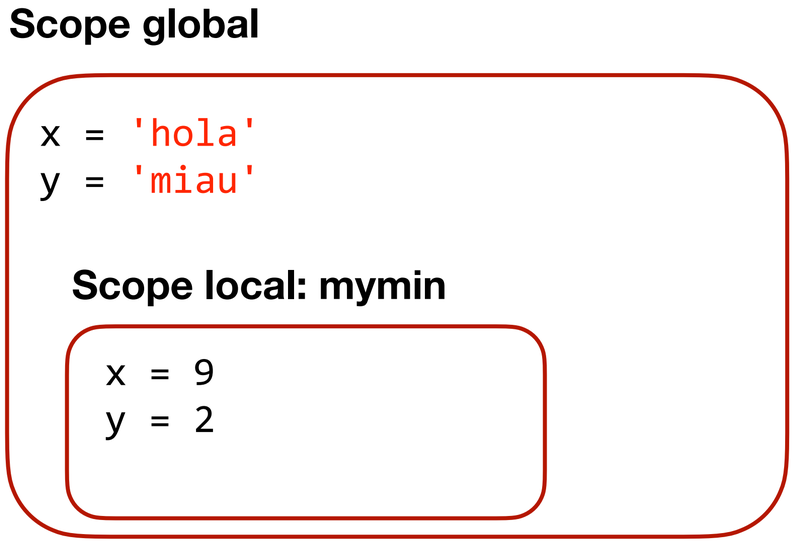
Además de los argumentos, hay lo que se llama **argumentos clave (keyword arguments)** que se especifican por nombre específicos. Uno de estos posibles argumentos con nombre para la función `print()` es `sep`, que le dice a `print()` qué caracter usar para separar los argumentos que va a imprimir cuando hay multiples elementos, por ejemplo:

```
print('a', 'b', 'c', sep='--')
```

Scope

- Scope (o ámbito) define la visibilidad de variables.
- El scope por defecto es el global. Todas las funciones tienen acceso al scope global.
- Cada función tiene su propio scope local, y solo se puede acceder a esas variables desde la función.
- Los parámetros de las funciones son parte del scope local.

```
 1 x = 'hola'
 2 y = 'miau'
 3
 4 def mymin(x, y):
 5     if x < y: return x
 6     else: return y
 7
 8 m = mymin(9,2)
 9
10 print (m)
```



Parámetros: alias o paso por referencia

- Los parámetros de una función son alias (referencias). Consecuencia: si variables son mutables, es posible modificar su contenido, aunque estén en otro scope!
- Tipos de datos inmutables: int, float, str, bool.
- Tipos de datos mutables: listas

```
1 def inc(L):
2     L.append(33)
3
4 a = [1, 2]
5 print(a)
6 inc(a)
7 print(a)
```

Modifica variable **a** en scope global

```
1 def inc(j):
2     j += 1
3
4 i = 99
5 print(i)
6 inc(i)
7 print(i)
```

No modifica **i**, pues **i** es inmutable

```
1 def inc(j):
2     j += 1
3     return j
4
5 i = 99
6 print(i)
7 i = inc(i)
8 print(i)
```

No modifica **i**, pues **i** es inmutable, pero, en línea 7 se modifica!

Algunos Trucos

```
def fibonacci(N):
    L = []
    a=0
    b=1
    while len(L) < N:
      c = a
      a = b
      b = c + b
        L.append(a)
    return L
```

=

```
def fibonacci(N):
    L = []
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L
```

Valores por omisión

A veces cuando definimos una función queremos darle al usuario de la función un poco más de flexibilidad con respecto a ciertos valores. En este caso, definimos valores por "omisión" de los argumentos. Consideremos la función **`fibonacci`** anterior. Cómo podríamos hacer que el usuario eligiera el valor de comienzo? Por ejemplo, así:

```
def fibonacci(N, a=0, b=1):
    L = []
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L
```

```
fibonacci(10)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

```
fibonacci(a=1, b=3)
[3, 4, 7, 11, 18, 29, 47, 76, 123, 199]
```

Funciones que retornan múltiples valores

Una función puede devolver (**`return`**) múltiples valores simplemente separando los valores por comas.

Ejemplo: números complejos:

```
def real_imag_conj(val):
    return val.real, val.imag, val.conjugate()

r, i, c = real_imag_conj(3 + 4j)
print(r, i , c)
```

Resumen

Conceptos

- **Función:** bloque de código que solo se activa cuando se ejecuta.
- **Scope:** ámbito en el que está disponible una variable

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

https://docs.python.org/3/reference/lexical_analysis.html

Funciones

- **min, max, round, sum:** obtiene mínimo, máximo, redondea al entero más cercano, suma.

		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

https://docs.python.org/3/library/functions.html