

PROGRAMANDO CON python™

Margarita Ruiz Olazar
mruizo@ucom.edu.py

Abril 2019

REFERENCIAS BIBLIOGRÁFICAS

- [Python 3.7.3 documentation](#)
- [Floating Point Arithmetic: Issues and Limitations](#)
- [Data Structures](#)
- Guttag, J. V. (2013). *Introduction to computation and programming using Python*. Mit Press.
- [Introduction to Computer Science and Programming in Python - MIT open courseware](#)
- [Python Data Science Handbook](#)
- [Learning Python: From Zero to Hero](#)
- [Medium corporation](#)
- Downey, A. B. (2012). *Think Python: How to Think Like a Computer Scientist, Version 2.0. 17*. Green Tea Press.
- [stackoverflow](#)



CONTENIDO

○ Clase 1

- Organizar y modularizar (Estructura del lenguaje Python)
- Manipulación de datos (Tipos de datos)
- Descomposición, abstracción y funciones
- Tuplas, listas, diccionarios

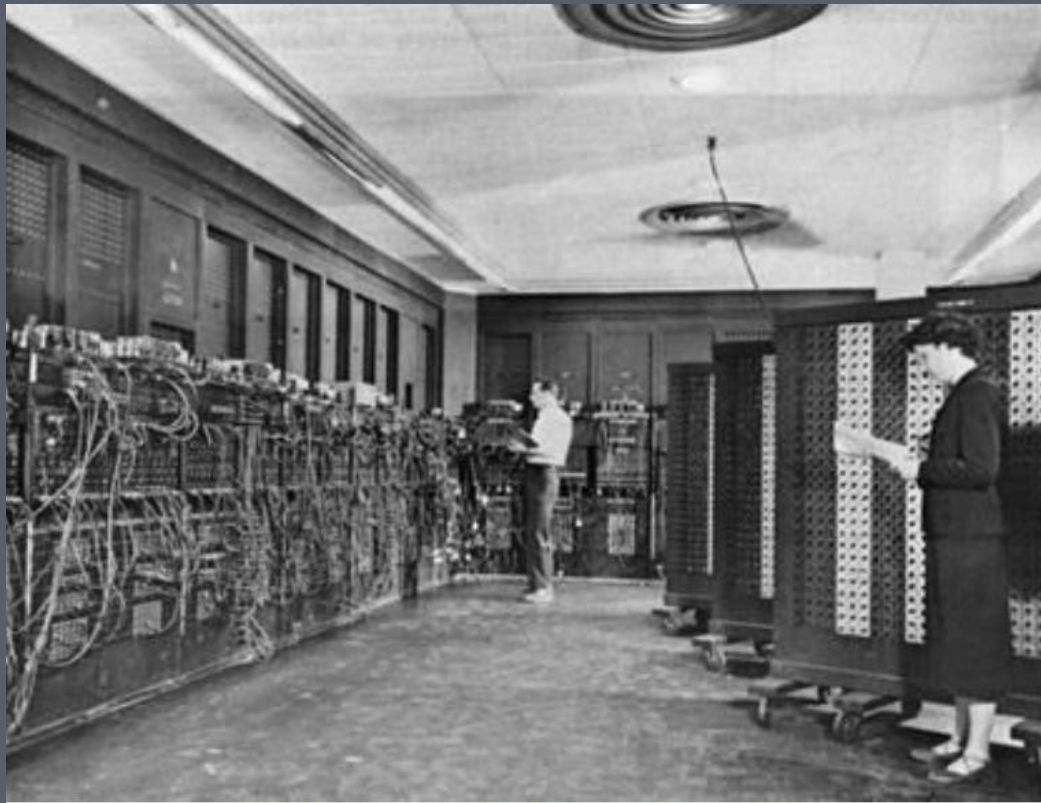
○ Clase 2

- aliasing, mutabilidad
- Recursión
- Depuración y excepciones
- Programación orientada a objetos (Clases y herencia)

○ Clase 3

- Virtual environment
- Paquetes y librerías Python
- Numpy, pandas, matplotlib
- Algoritmos eficientes.
- Algoritmos de búsqueda y ordenación.





Primera computadora del mundo

CLASE 1

LENGUAJE DE PROGRAMACIÓN

- Es una forma estandarizada para comunicar comandos a un computador, o sea, a través de diversas reglas semánticas y sintácticas, podemos instruir el computador a hacer algo.



PARADIGMA

- Es un modelo o un padrón, luego un paradigma de programación es una forma de estructurar el código. Un paradigma de programación también determina la visión que el desarrollador tiene sobre la resolución de un problema y ejecución de un programa.
- En un primer nivel los paradigmas se pueden clasificar en función de la aproximación que adoptan para la solución del problema.



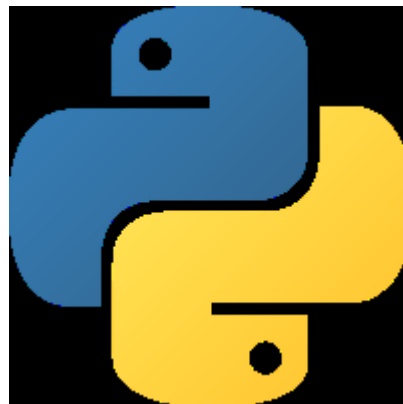
ASPECTOS DE LOS LENGUAJES

- Construcciones primitivas
 - Palabras en inglés
 - Lenguaje de programación: números, cadenas, operadores simples.
- Sintaxis
 - Lenguaje de programación: una instrucción puede tener o no tener validez sintáctica.
- Semántica
 - La semántica estática es cuando los *strings* sintácticamente válidos tienen un significado.
 - Ej. $3 + \text{"hola"}$ -> error semántico estático
 - La semántica es el significado asociado con una serie de símbolos sintácticamente correctos sin errores semánticos estáticos



OBJETIVO: PROGRAMAR EN PYTHON

- Aprender la sintaxis y la semántica de un lenguaje de programación.
- Aprender cómo usar estos elementos para resolver un problema en una forma que la computadora pueda hacer el trabajo por nosotros.
- Aprender los modos computacionales de pensamiento para permitirnos aprovechar un conjunto de métodos para resolver problemas complejos.



Lanzado en 1991, desarrollado inicialmente por Guido van Rossum



PROGRAMAR EN PYTHON

- **Python** es un lenguaje de **programación** interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible.
- Se trata de un lenguaje de **programación** multi-paradigma, ya que soporta orientación a objetos, **programación** imperativa y, en menor medida, **programación** funcional.



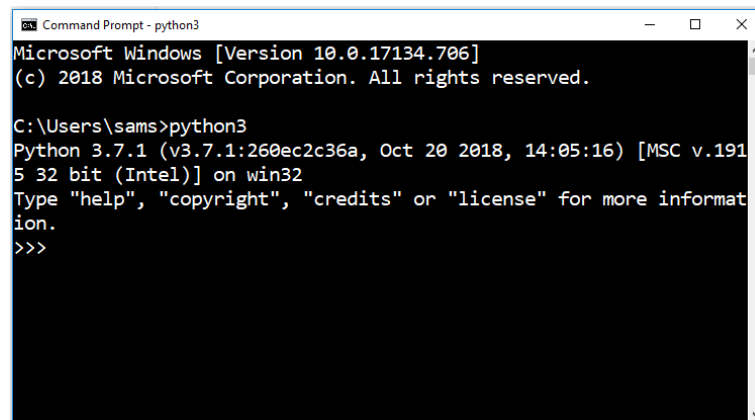
“high-level programming language, and its core design philosophy is all about code readability and a syntax which allows programmers to express concepts in a few lines of code.”

Guido van Rossum



PROGRAMAR EN PYTHON

- un programa es una secuencia de definiciones y comandos.
- el interprete de Python evalúa cada secuencia.
- los comandos son ejecutados por el interprete de python en un *shell*
- los comandos (instrucciones) instruyen al intérprete para que haga algo.



```
Command Prompt - python3
Microsoft Windows [Version 10.0.17134.706]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\sams>python3
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.191
5 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more informat
ion.
>>>
```

Puede ser directamente escrito en un *shell* o almacenado en un archivo que es leído en el *shell* y evaluado.



OBJETOS

- programas manipulan objetos de datos
- los objetos tienen un **tipo** que define el tipo de operaciones que los programas pueden hacer sobre estos.
- los objetos son:
 - escalar (no puede ser subdividido)
 - no-escalar (tiene una estructura interna a la que se puede acceder)

```
struct _longobject {  
    long ob_refcnt;  
    PyTypeObject *ob_type;  
    size_t ob_size;  
    long ob_digit[1];  
};
```

Representación de un número entero en Python 3.4 (lenguaje C)



OBJETOS ESCALARES

- `int` - representa **enteros**, ej. 5
- `float` - representa **números reales**, ej. 3.27
- `Bool`- representa valores **booleanos** verdaderos y falsos
- `NoneType`: **special** y tiene un valor, Ninguno
- puede usar `type ()` para ver el tipo de un objeto

```
In [1] : type(5)
```

Lo que escribes en el shell de Python

```
Out[1] : int
```

Lo que muestra después de pulsar Enter

```
In [2] : type(3.0)
```

```
Out[2] : float
```



CONVERSIÓN DE TIPO (CAST)

- Puede **convertir objetos de un tipo a otro**
- `float(3)` convierte el entero 3 a flotante 3.0
- `int(3.9)` trunca flotante 3.9 a entero 3



IMPRIMIR A LA CONSOLA

- Para mostrar la salida de código en la consola, use el comando `print`

```
In [11] : 3+2
```

```
Out[11] : 5
```

```
In [12] : print(3+2)
```

```
5
```



EXPRESIONES

- Combinar objetos y operadores para formar expresiones
- Una expresión tiene un **valor**, el cual tiene un tipo
- La sintaxis para un expresión simple

`<object> <operator> <object>`



OPERADORES SOBRE INT'S Y FLOAT'S

- $i + j$ → suma
- $i - j$ → diferencia
- $i * j$ → producto
- i / j → división
- $i // j$ → int división
- $i \% j$ → el **resto** cuando i es dividido por j
- $i ** j$ → i a la **potencia** de j



OPERACIONES SIMPLES

- Paréntesis usados para decirle a Python el orden de ejecución de las operaciones
 - $3*5+1$ evalúa 16
 - $3*(5+1)$ evalúa 18
- Operadores de precedencia cuando no se usan paréntesis
 - $**$
 - $*$
 - $/$
 - $+$ y $-$ ejecutado de izquierda a derecha, como aparece en la expresión



OPERADORES DE REDONDEO

- Python tiene una función `round()` que toma dos argumentos numéricos, `n` y `ndigits`, y devuelve el número `n` redondeado a `ndigits`.
- el argumento `ndigits` se establece en cero de manera predeterminada, por lo que omitirlo da como resultado un número redondeado a un entero.
- es posible que `round()` no funcione como se espera.

```
>>> round(2.5)
2
>>> round(1.5)
2
```

- error de representación de punto flotante



ERROR DE REPRESENTACIÓN DE PUNTO FLOTANTE

- es un problema causado por la representación interna de números de punto flotante,
- el computador utiliza un número fijo de dígitos binarios para representar un número decimal.
- algunos números decimales no se pueden representar exactamente en binario, lo que resulta en pequeños errores de redondeo.

```
>>> 1 - .8
0.19999999999999996
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```



OPERADORES DE INPUT/OUTPUT

○ print

```
x = 1  
print(x)
```

```
x_str= str(x)      Convierte un entero a un string  
print("mi fav num es", x, ".", "x =", x)  
print("mi fav num es "+ x_str + ". "+ "x = "+ x_str)
```

○ input

```
text = input("ingrese algo... ")  
print(5*text)
```

Se puede asociar la
entrada a una variable

```
num= int(input("ingrese un número... "))  
print(5*num)
```

Retorna un string, para
trabajar con números
se debe convertir



VARIABLES Y VALORES

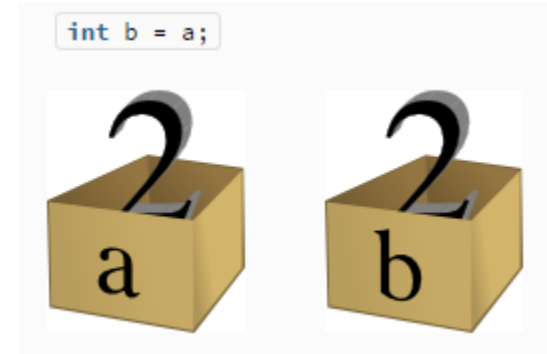
- El signo igual es la **asignación** de un valor a un nombre de variable.

- $\boxed{\text{pi}} = \boxed{3.14159}$
variable *valor*
- $\text{pi_approx} = \boxed{22/7}$
el valor de la expresión es 3

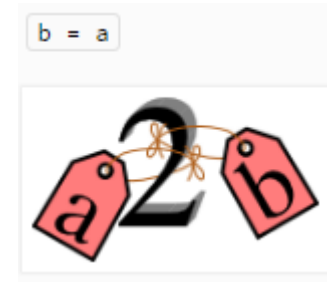
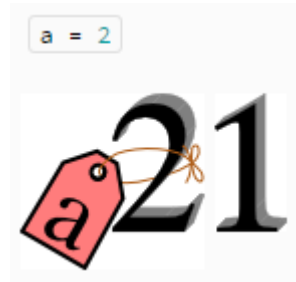
- El valor es almacenado en la memoria de la computadora.
- Una asignación asocia el nombre al valor
- Se recupera el valor asociado con el nombre o variable, invocando el nombre. Ej. escribiendo `pi`



ASOCIACIÓN DE VALORES A VARIABLES



Lenguajes tipados



Lenguaje Python



EXPRESIONES ABSTRACTAS

- Porqué damos nombre a los valores de expresiones?
 - Para reusar nombres en vez de valores
 - Mas fácil cambiar el código posteriormente

```
pi = 3.14159  
radius = 2.2  
area = pi*(radius**2)
```



OPERADORES DE COMPARACIÓN (INT Y FLOAT)

Para las variables i y j

- $i > j$ retorna **True** (verdadero) si i es mayor que j
- $i \geq j$ retorna **True** (verdadero) si i es mayor o igual que j
- $i < j$ retorna **True** (verdadero) si i es menor que j
- $i \leq j$ retorna **True** (verdadero) si i es menor o igual que j
- $i == j$ test de **igualdad**. Retorna **True** si i es igual a j
- $i != j$ test de **desigualdad**. Retorna **True** si i no es igual a j



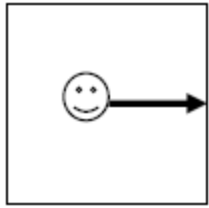
OPERADORES LÓGICOS (BOOL)

Sean a y b variables y/o expresiones

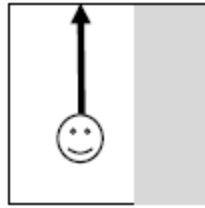
- $\text{not } a$ retorna **True** si a es **False**. **False** si a es **True**
- $a \text{ and } b$ retorna **True** si ambos son **True**
- $a \text{ or } b$ retorna **True** si alguno o ambos son **True**

a	b	$a \text{ and } b$	$a \text{ or } b$
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

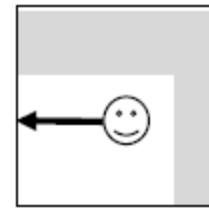




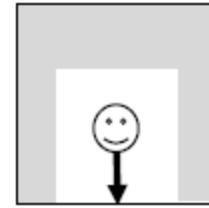
Si esta libre a la derecha,
Ir a la derecha



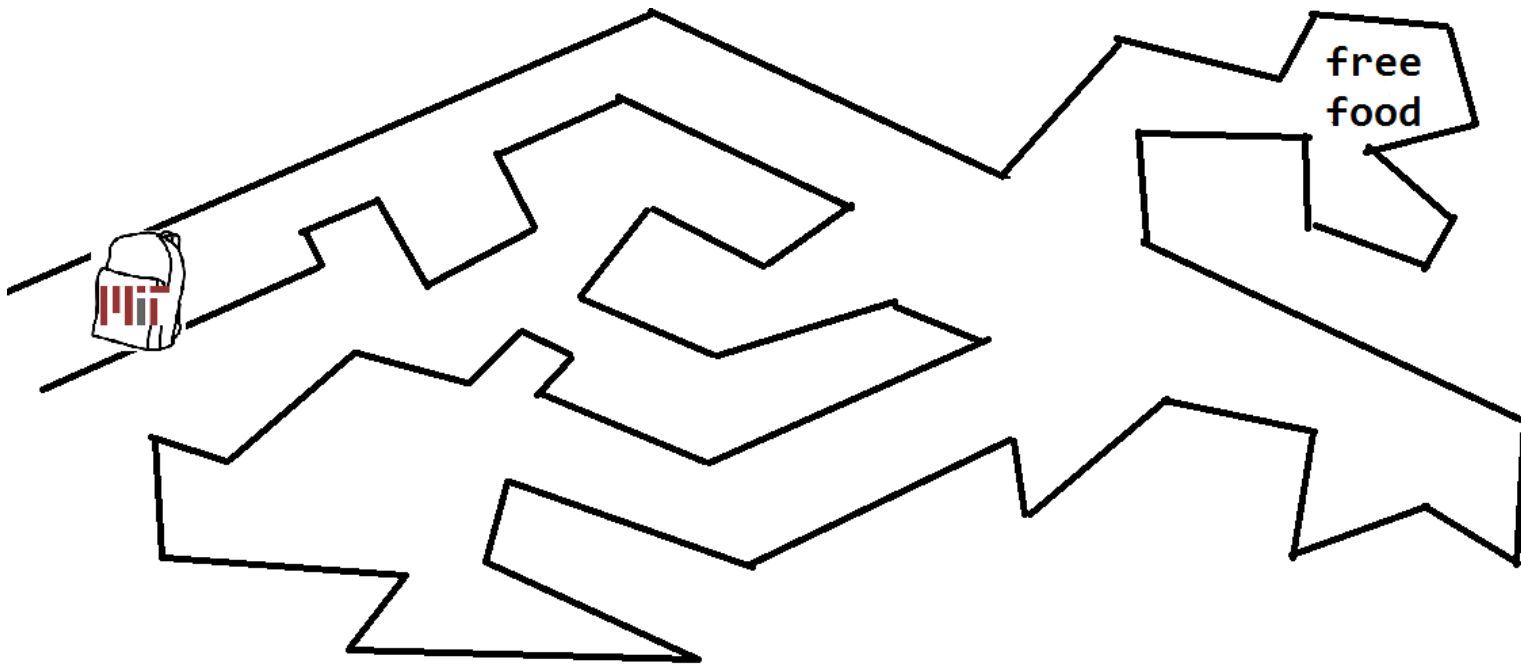
Si está bloqueado a la derecha,
Ir hacia adelante



Si a la derecha y enfrente están
bloqueados,
Ir a la izquierda

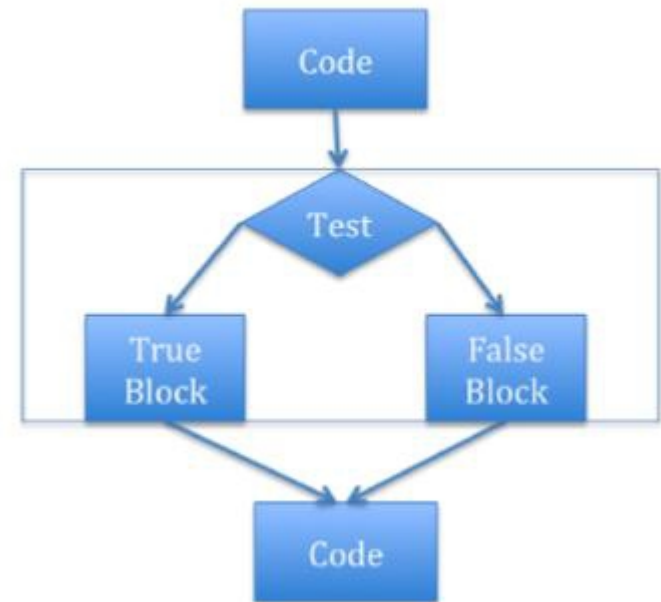


Si a la derecha, enfrente,
izquierda bloqueados,
Volver para atrás



BIFURCACIÓN DE PROGRAMAS

- La instrucción de bifurcación más simple es un condicional.
- Una condición que se evalúa como Verdadero o Falso.
- Un bloque de código para ejecutar si la condición es verdadera.
- Un bloque de código opcional para ejecutar si la condición es Falsa.



CONTROL DE FLUJO - BIFURCACIÓN

```
if <condition>:  
    <expression>  
    <expression>  
    ...
```

```
if <condition>:  
    <expression>  
    <expression>  
    ...  
else:  
    <expression>  
    <expression>  
    ...
```

```
if <condition>:  
    <expression>  
    <expression>  
    ...  
elif <condition>:  
    <expression>  
    <expression>  
    ...  
else:  
    <expression>  
    <expression>  
    ...
```

<condition> tiene valor True o False

Ejecuta las expresiones en el bloque donde if<condition> es True



INDENTACIÓN

- Importa en Python
- Denota los bloques de código

```
x = float(input("Enter a number for x: "))
y = float(input("Enter a number for y: "))
if x == y:
    print("x and y are equal")
    if y != 0:
        print("therefore, x / y is", x/y)
elif x < y:
    print("x is smaller")
else:
    print("y is smaller")
print("thanks!")
```



EJERCICIO

1. Escriba un programa en python para evaluar si un número entero es par o impar. Ingrese el número por consola.
2. Escriba un programa en python para evaluar si un número entero es divisible por 2 y por 3. O divisible por 2 pero no por 3. O si el número es divisible por 3 y no por 2. Ingrese el número por consola.
3. Escriba un programa en Python que reciba 3 números enteros por consola y evalúe cuál es menor.



EJEMPLO 1

```
x= int(input('Ingrese un entero: '))  
if x%2 == 0:  
    print(''  
    print('El número es Par')  
else:  
    print(''  
    print('El número es Impar')  
print('Realizado con condicional')
```



EJEMPLO 2

```
x= int(input('Ingrese un entero: '))  
if x%2 == 0:  
    if x%3 == 0:  
        print('Divisible por 2 y 3')  
    else:  
        print('Divisible por 2 y no por 3')  
elif x%3 == 0:  
    print('Divisible por 3 y no por 2')
```



INCLUYENDO BOOLEANOS

```
x= int(input('Ingrese un entero: '))
y= int(input('Ingrese un entero: '))
z= int(input('Ingrese un entero: '))

if x < y and x < z:
    print('x es menor')
elif y < z:
    print('y es menor')
else:
    print('z es menor')
```



BIFURCACIONES

- Los programas de bifurcación nos permiten tomar decisiones y hacer cosas diferentes.
- Pero aún así, como máximo, cada instrucción se ejecuta una vez.
- El tiempo máximo para ejecutar el programa depende solo de la duración del programa
- Estos programas se ejecutan en **tiempo constante**.



TIPOS DE DATOS

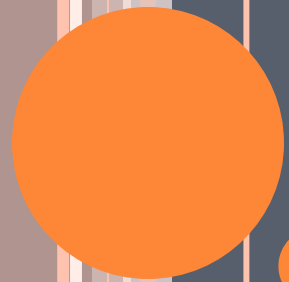
- Variables y expresiones
 - int
 - float
 - bool
 - `string`
 - otros



STRINGS O CADENAS DE CARACTERES

- Letras, caracteres especiales, espacios, dígitos
- En Python los *strings* deben estar encerrados entre "comillas dobles" o 'apóstrofes'
 - `hola = "hola a todos"`
 - `saludo = 'hola'`
- También podemos concatenar *strings*
 - `nombre = "Ana"`
 - `saludar = saludo + nombre`
 - `saludos = saludo + " " + nombre`
- La función `len()` es usada para obtener la longitud del *string*.
 - `s = "abc"`
 - `len(s) -> retorna 3`





STRING SLICING

STRING SLICING (1)

- Indexación y partición de cadenas de caracteres son herramientas increíbles en programación.
- Podemos pensar que estas cadenas pueden ser divididas en una grilla y cada caracter en la cadena tiene su propio lugar en la grilla.
- Este lugar es el **índice**, definido por un entero que va de **cero** en adelante para tantos caracteres como contiene la cadena.

a = " U c o m 2 0 1 9 "

U	c	o	m		2	0	1	9
---	---	---	---	--	---	---	---	---

↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑

[0] [1] [2] [3] [4] [5] [6] [7] [8]




STRING SLICING (2)

- Que hago si necesito trabajar con un pequeño subconjunto del *String*?

a = “

U	c	o	m		2	0	1	9
---	---	---	---	--	---	---	---	---

 ”



subString = a[inicio:fin:paso]

Índice del primer carácter que queremos incluir en nuestro substring

Índice del último carácter en el substring. No será incluido. El carácter inmediatamente a su izquierda será el último carácter.

Indica como saltar de un índice a otro. Determina cual carácter incluir.



DIFERENTES ALTERNATIVAS USANDO STRING SLICING (1)

- Si colocamos un número entre [].

Ej. `a[3] = 'm'` obtendremos el caracter en ese índice.

- Si colocamos `:` antes del número.

Ej. `a[:3] = 'Uco'` obtendremos la parte del *string* que inicia en el índice 0 hasta el carácter localizado a la izquierda del índice 3.

- Si colocamos `:` después del número.

Ej. `a[3:] = 'm 2019'` obtendremos la parte del *string* que inicia en el índice 3 hasta el carácter localizado en el final del *string*.

`a = “`

U	c	o	m		2	0	1	9
---	---	---	---	--	---	---	---	---

`”`

[0] [1] [2] [3] [4] [5] [6] [7] [8]



DIFERENTES ALTERNATIVAS USANDO STRING SLICING (2)

- Si colocamos dos números separados por `:` como:

Ej. [(número1) : (número2)], obtendremos la partición del string que comienza en número1 y contiene todos los caracteres en adelante hasta número2, pero no incluye el localizado en número2. Ej. `a[2:6] = 'om 2'`

- Si incluimos dos veces `:` antes de un número.

Ej. `a[::2] = 'uo 09'`, ese número será el salto desde el índice 0 al próximo índice hasta el final del string.

`a = " U c o m 2 0 1 9 "`

U	c	o	m		2	0	1	9
---	---	---	---	--	---	---	---	---

[0] [1] [2] [3] [4] [5] [6] [7] [8]

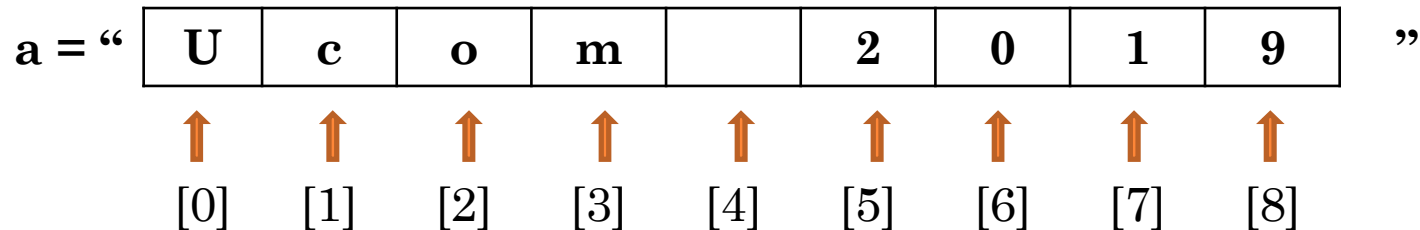


DIFERENTES ALTERNATIVAS USANDO STRING SLICING (3)

- Si colocamos tres números separados por `:` como:

Ej. [(número1) : (número2) : (número3)], obtendremos la partición del *string* que comienza en número1 y contiene todos los caracteres en adelante hasta número2, pero no incluye el localizado en número2. El tercer número indica los caracteres que serán incluidos.

Ej. `a[2:10:2] = 'o 09'` determina el sub-*string* que comienza en el índice 2, incluyendo los caracteres saltando cada 2 índices. Incluye los índices 2, 4, 6 y 8.



subString = `a[3: 9: 2]`



VALORES POR DEFECTO

Si no especificamos:

- El índice **inicial**, por defecto es [0] el primer carácter.
- El índice **final**, por defecto es el último carácter.
- El **paso**, por defecto es 1 (incluye cada carácter).

```
a[::step] #From start to end using this step
```

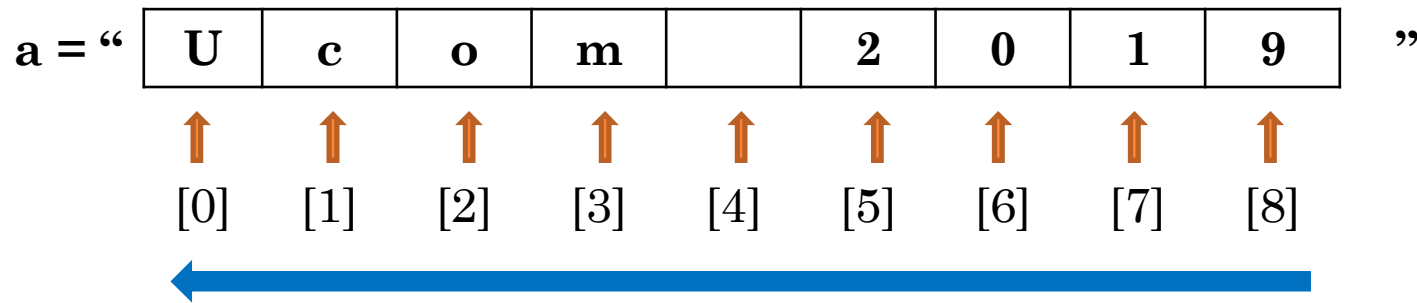
```
a[start::step] #From start index to end character using step
```

```
a[:end:step] #From first character to end index using step
```

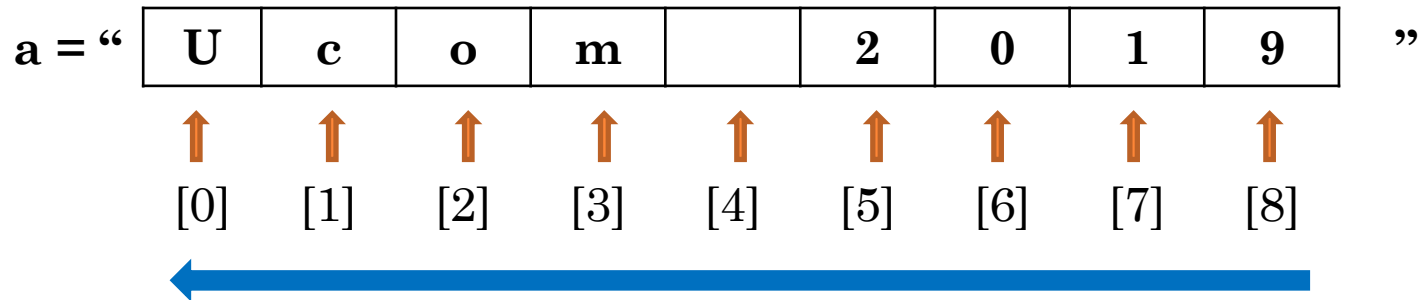


PASO CON VALOR NEGATIVO

- Cuando usamos pasos negativos estamos determinando que recorreremos el string de derecha a izquierda.



subString = a[: : -1] = '9102 mocu'



subString = a[8: 2: -2] = '90 '



COMPARANDO *STRINGS*

- Una secuencia de objetos puede ser comparada a otros objetos del mismo tipo.

'ABCI' > 'ABCA'

1. 'A'=='A' ? True
2. 'B'=='B' ? True
3. 'C'=='C' ? True
4. 'I'=='A' ? False
5. 'I' > 'A' ? True

?

- La comparación de *strings* usa ordenamiento lexicográfico.

El orden lexicográfico usa el *Unicode code point number* para ordenar caracteres



ORDEN LEXICOGRÁFICO

Python usa la función `ord()` para obtener el orden lexicográfico de un carácter.

`'JKA' > 'JKL'`

Character from left string	Corresponding Unicode	Character from right string	Corresponding Unicode	Unicode comparison
'J'	<pre>>>> ord('J') 74 .</pre>	'J'	<pre>>>> ord('J') 74 .</pre>	74==74 True
'K'	<pre>>>> ord('K') 75 .</pre>	'K'	<pre>>>> ord('K') 75 .</pre>	75==75 True
'A'	<pre>>>> ord('A') 65 .</pre>	'L'	<pre>>>> ord('L') 76 .</pre>	65==76 False

65 > 76

False

`'JKA' > 'JKL'` is False

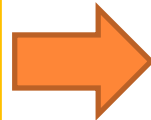
```
>>> 'JKA' > 'JKL'
False
```

IMPORTANTE: el Unicode de una letra en mayúsculas y minúsculas no son lo mismo!

TRABAJANDO CON *STRINGS* (1)

- En Python un caracter es un *string* de longitud 1.
- Si incluimos “ o ‘ dentro de un *string*, Python considerará esto muy confuso y el interprete puede evaluarlo como un error. Que hacemos?

```
>>> foo = "comillas (") son mejores"
File "<stdin>", line 1
    foo = "comillas (") son mejores"
                ^
SyntaxError: invalid syntax
>>> 'That's life'
File "<stdin>", line 1
    'That's life'
        ^
SyntaxError: invalid syntax
>>> bar = 'That's life'
File "<stdin>", line 1
    bar = 'That's life'
            ^
SyntaxError: invalid syntax
```



```
>>> foo = 'comillas (") son mejores'
>>> bar = "That's life"
>>> print(foo)
comillas (") son mejores
>>> print(bar)
That's life
```

```
>>> foo = "comillas (\") son mejores"
>>> bar = 'That\'s life'
```



TRABAJANDO CON *STRINGS* (2)

Escape	What it does.
<code>\\</code>	Backslash (<code>\</code>)
<code>\'</code>	Single-quote (<code>'</code>)
<code>\"</code>	Double-quote (<code>"</code>)
<code>\a</code>	ASCII bell (BEL)
<code>\b</code>	ASCII backspace (BS)
<code>\f</code>	ASCII formfeed (FF)
<code>\n</code>	ASCII linefeed (LF)
<code>\N{name}</code>	Character named name in the Unicode database (Unicode only)
<code>\r ASCII</code>	Carriage Return (CR)
<code>\t ASCII</code>	Horizontal Tab (TAB)
<code>\uxxxx</code>	Character with 16-bit hex value xxxx (Unicode only)
<code>\Uxxxxxxxx</code>	Character with 32-bit hex value xxxxxxxx (Unicode only)
<code>\v</code>	ASCII vertical tab (VT)
<code>\ooo</code>	Character with octal value ooo
<code>\xhh</code>	Character with hex value hh

Tabla1: caracteres especiales que pueden ser usados en Python



SÍMBOLOS DE OPERADORES

- Los más comunes son: +, *, % .Tienen significados diferentes cuando se usan con *strings*

- + operador de concatenación

```
>>> foo = "Hola"
>>> var = "Mundo"
>>> foovar = foo + var
>>> print(foovar)
HolaMundo
```

- * operador de sobrecarga

```
>>> "abba" * 3
'abbaabbaabba'
```

```
>>> 'A' * 0
''
>>> 'b' * -1
''
```

- % operador de inserción

```
>>> celsius = 9
>>> print("%i C = %.2f F"% (celsius, celsius * 1.8 + 32))
9 C = 48.20 F
```

- Triple comillas para comentarios de varias líneas

```
""" """ or ''' '''
```

OPERADOR IN Y NOT IN

- Los operadores `in` y `not in` verifican la pertenencia de los miembros de una colección.
 - elemento en colección
- Retorna **True** si el elemento es un miembro de la colección y **False** en caso contrario.



USANDO CONTROLES EN *LOOPS*

- Los programas de bifurcación simplemente toman decisiones, pero el camino a través del código todavía es lineal.
- En ocasiones, se desea reutilizar partes del código un número indeterminado de veces.



LOOP CON WHILE

```
while <condition>:  
    <expression>  
    <expression>  
    ...
```

- <condition> retorna un Boolean
- Si la evaluación de <condition> es True, se realizan todos los pasos dentro del bloque de código while
- A continuación se evalúa nuevamente <condition>
- Se repite hasta que <condition> sea False



LOOP CON FOR

```
for <variable> in range(<some_num>) :  
    <expression>  
    <expression>  
    ...
```

- <variable> es un índice para recorrer la colección
- cada vez que se evalúa el loop, <variable> toma un valor.
- la primera vez, <variable> inicia con un valor específico.
- la siguiente vez, <variable> es incrementada en 1
- se repite hasta recorrer toda la colección



LOOPS WHILE Y FOR

- Iterar a través de una secuencia de números.

más complicado con while

```
n = 0
while n < 5:
    print(n)
    n = n+1
```

mas corto con for

```
for n in range(5):
    print(n)
```

Operadores aritméticos abreviados

`+=` , `-=` , `*=` , `/=`



PYTHONTUTOR

<http://www.pythontutor.com/>

Vale la pena!

VISUALIZE CODE AND GET LIVE HELP

Learn Python, Java, C, C++, JavaScript, and Ruby

Python Tutor, created by [Philip Guo](#) ([@pgbovine](#)), helps people overcome a fundamental barrier to learning programming: understanding what happens as the computer runs each line of code.

Write code in your web browser, see it visualized step by step, and get live help from volunteers.

Related services: [Java Tutor](#), [C Tutor](#), [C++ Tutor](#), [JavaScript Tutor](#), [Ruby Tutor](#)

So far, **over five million people in over 180 countries** have used Python Tutor to visualize over 75 million pieces of code, often as a supplement to textbooks, lectures, and online tutorials.

[Visualize your code and get live help now](#)

The screenshot displays the Python Tutor interface. On the left, a '1-minute introduction' section lists users who need help with Python, such as 'user_116' and 'user_321'. Below this, a code editor shows Python 2.7 code:

```
1 x = ['a', 'b', 'c']
2 y = ['d', 'e', 'f']
3 z = x
```

. To the right of the code, a 'Frames' and 'Objects' diagram visualizes the memory state. The 'Global frame' contains variables x, y, and z. The 'Objects' section shows the list objects for x and y, which both point to the same memory location containing ['a', 'b', 'c']. The 'z' variable points to a new memory location containing ['d', 'e', 'f']. On the right side of the interface, a 'Thank you for helping!' message is displayed, along with a 'Generate permanent link' button and a 'Generate shortened link' button. A 'Program terminated' message is also visible.

RANGE(INICIO, FIN, PASO)

- Los valores por defecto son inicio = 0 y paso = 1 y es opcional.
- Se detiene cuando fin es fin-1

```
mysum= 0
for i in range(7, 10):
    mysum += i
    print(mysum)
mysum= 0
```

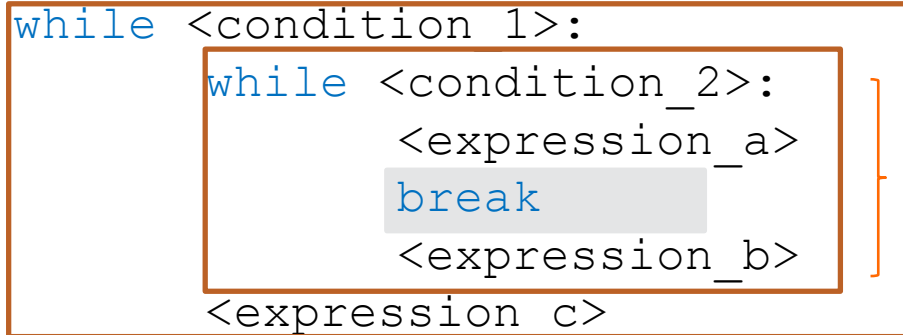
```
for i in range(5, 11, 2):
    mysum += i
    print(mysum)
```



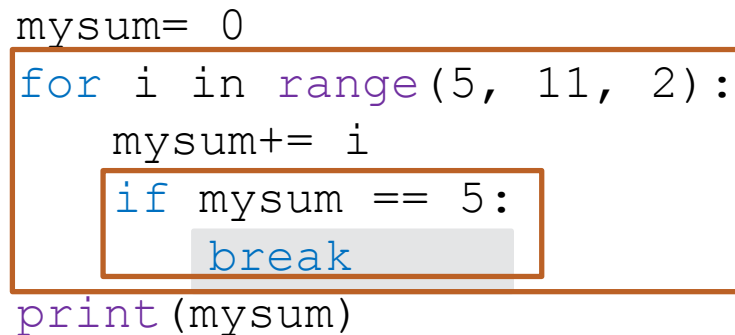
DECLARACIÓN BREAK

- Abandona inmediatamente cualquier **loop** dentro del cual se encuentra.

```
while <condition 1>:  
    while <condition_2>:  
        <expression_a>  
        break  
        <expression_b>  
    <expression c>
```



```
mysum= 0  
for i in range(5, 11, 2):  
    mysum+= i  
    if mysum == 5:  
        break  
print(mysum)
```



FOR VS WHILE

for

- conoce el número de iteraciones
- puede terminar antes con un `break`
- se puede escribir un loop `for` usando `while`
- usa un contador

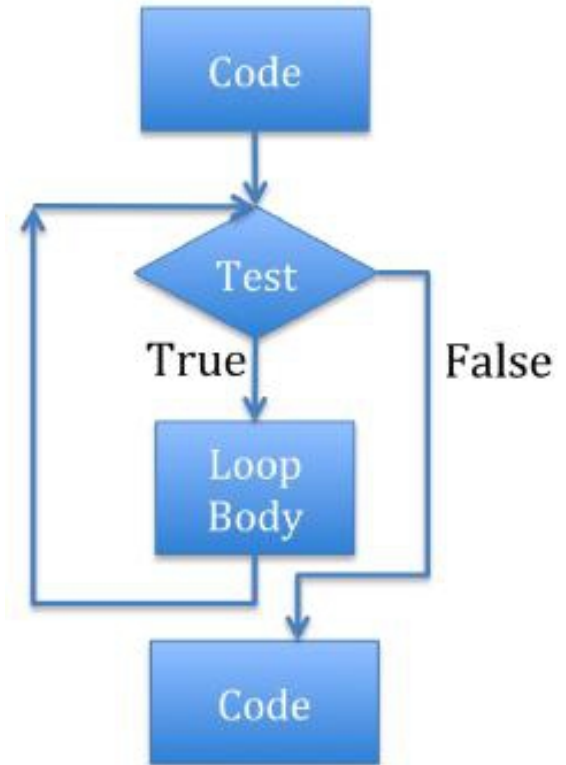
while

- el número de iteraciones es ilimitadas
- puede terminar antes usando un `break`
- no se puede escribir un loop `while` usando `for`
- se puede usar un contador pero se debe inicializar antes del loop e incrementar/decrementar dentro del loop



ITERACIÓN

- El concepto de iteración nos permite extender algoritmos de bifurcación para poder escribir programas de complejidad arbitraria
- Se inicia con un test
- Si se evalúa como `True`, luego se ejecuta el cuerpo del bucle una vez, y retorna para evaluar nuevamente el test
- Esto se repite hasta que el test se evalúe como `False`, seguidamente se ejecuta el código que sigue a la instrucción de iteración



EJEMPLO

```
x = 3
res = 0
iteraIzq = x
while(iteraIzq != 0):
    res = res + x
    iteraIzq = iteraIzq - 1
print(str(x) + '*' + str(x) + ' = ' + str(res))
```

El código ajusta el valor de x^2 por adición repetitiva

x	res (x^2)	iteraIzq
3	0	3
	3	2
	6	1
	9	0



CLASES DE ALGORITMOS

- Los algoritmos iterativos nos permiten hacer cosas más complejas que la aritmética simple.
- Podemos repetir una secuencia de pasos varias veces en función de alguna decisión; conduce a nuevas clases de algoritmos.
- Un ejemplo útil son los métodos de “suponer y verificar”



SUPONER Y VERIFICAR

- Si pudiéramos suponer valores posibles para la raíz cuadrada (llámela g), entonces podemos usar la definición para verificar si $g * g = x$
- Sólo necesitamos una buena manera de generar suposiciones.

Ejemplo: Encontrar la raíz cúbica de un número entero

- Una forma de utilizar esta idea de generar suposiciones para encontrar una raíz cúbica de x es intentar primero $0^{**}3$, luego $1^{**}3$, luego $2^{**}3$, etc.
- Puede detenerse cuando alcance k tal que $k^{**}3 > x$
- Existe un número finito de casos para tratar





EJERCICIOS

Iteración, control de flujo, trabajando con strings

CÓMO ESCRIBIR CÓDIGO?

- más código no es necesariamente algo bueno
- buenos programadores se miden por la cantidad de funcionalidad
- utilizar mecanismos para lograr la descomposición y abstracción.
 - **ÍDEA DE ABSTRACCIÓN**: no necesitamos saber cómo el proyector funciona para usarlo.
 - **IDEA de DECOMPOSICION**: diferentes dispositivos trabajan juntos para lograr un objetivo final

Juntos son poderosos

El código puede ser usado muchas veces
pero solo tiene que ser depurado una vez!



FUNCIONES

- **Funciones**: piezas/trozos de código reutilizables, llamadas **funciones**
- las funciones no se ejecutan en un programa hasta que son "**llamadas**" o "**invocadas**" en un programa
- características de la función:
 - tiene un nombre
 - tiene parámetros (0 o más)
 - tiene un *docstring* (opcional pero recomendado)
 - tiene un cuerpo
 - devuelve algo



COMO ESCRIBIR Y LLAMAR/INVOCAR A UNA FUNCIÓN EN PYTHON

Palabra clave

nombre

Parámetros o
argumentos

especificación
docstring

contrato

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("inside is_even")
    return i%2 == 0
```

cuerpo

Posteriormente en el código, se
llama la función usando su nombre
y los valores de los parámetros

```
is_even(3)
```



ÁMBITO DE UNA VARIABLE (SCOPE)

- El **parámetro formal** se vincula al valor del parámetro real cuando se llama a la función
- nuevo **ámbito/marco/entorno** creado cuando se ingresa una función
- *scope* es el mapeo de los nombres a objetos

```
def f(x):  
    x = x + 1  
    print('in f(x): x =', x)  
    return x
```

Definición de
la función

```
x = 3
```

```
z = f(x)
```

Parámetro
actual

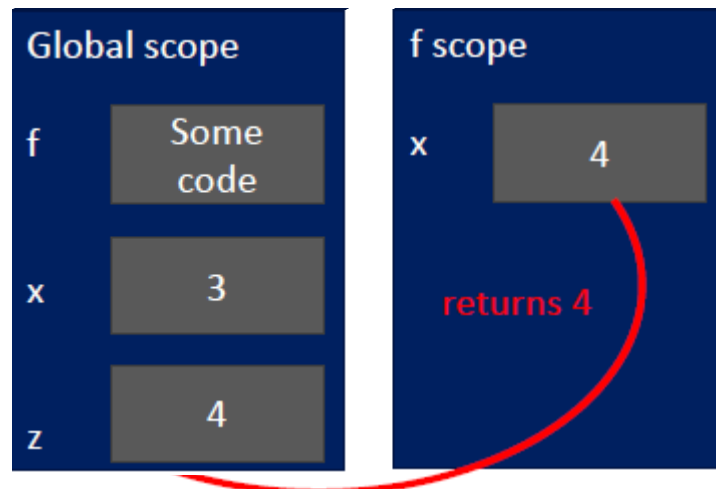
Programa principal (main)
* inicializa una variable x
* realiza la llamada a la f(x)
* asigna el valor de retorno de la
función a la variable z



ÁMBITO DE UNA VARIABLE (SCOPE)

```
def f( x ) :  
    x = x + 1  
    print('in f(x): x =', x)  
    return x
```

```
x = 3  
z = f( x )
```



SI NO TIENE RETORNO

- python retorna el valor `none`, si ningún retorno es determinado
- representa la ausencia de valor

```
def es_par( i ):  
    """  
    Entrada: i, un int positivo  
    No retorna nada  
    """  
    i%2 == 0
```



RETURN VS PRINT

- `return` solo tiene significado en la función;
- sólo existe un `return` en una función;
- el código dentro de la función no es ejecutada después de la declaración de `return`;
- tiene un valor asociado a él, el que es dado a la función que lo llama.
- `print` puede ser usada fuera de las funciones;
- se pueden ejecutar muchas instrucciones `print` dentro de una función;
- el código dentro de la función se puede ejecutar después de una declaración `print`;
- tiene un valor asociado a él, salida a la consola.



FUNCIONES COMO ARGUMENTOS

```
def func_a():  
    print('en la func_a')  
  
def func_b(y):  
    print ('en la func_b')  
    return y  
  
def func_c(z):  
    print ('en la func_c')  
    return z()
```

```
print(func_a())  
print(5 + func_b(2))  
print(func_c(func_a()))
```

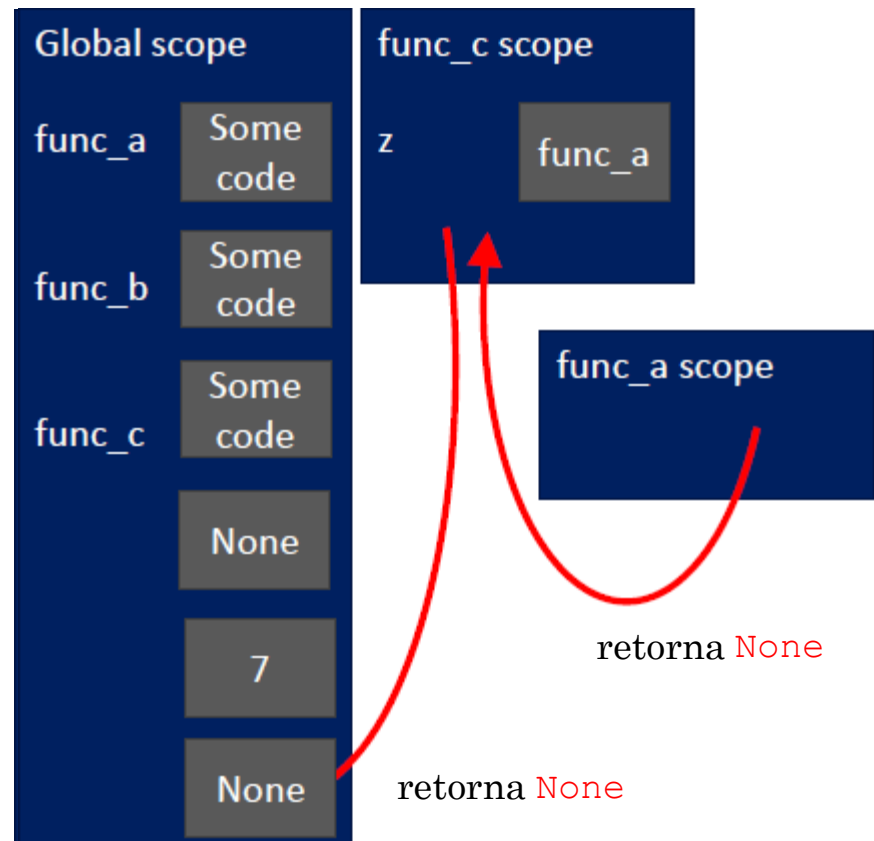
func_a no tiene parámetros
func_b tiene un parámetro
*func_c tiene un parámetro,
otra función*



FUNCIONES COMO ARGUMENTOS

Podemos tener argumentos de cualquier tipo, también funciones

```
def func_a():  
    print('en la func_a')  
  
def func_b(y):  
    print('en la func_b')  
    return y  
  
def func_c(z):  
    print('en la func_c')  
    return z()  
  
print(func_a())  
print(5 + func_b(2))  
print(func_c(func_a))
```



ARGUMENTOS Y VALORES POR DEFECTO

- Cada una de estas invocaciones son equivalentes:

```
printName('Ana', 'Gómez', False)
```

```
printName('Ana', 'Gómez', reverse = False)
```

```
printName('Ana', lastName= 'Gómez', reverse = False)
```

```
printName(lastName= 'Gómez', firstName= 'Ana', reverse = False)
```



ARGUMENTOS Y VALORES POR DEFECTO

- Definición de función simple: si el último argumento es **TRUE**, imprima apellido, nombre; en caso contrario imprima nombre, apellido

```
def printName(firstName, lastName, reverse):  
    if reverse:  
        print(lastName+ ', ' + firstName)  
    else:  
        print(firstName, lastName)
```



ARGUMENTOS Y VALORES POR DEFECTO

- Puede especificar que algunos argumentos tengan valores predeterminados, si no se proporciona ningún valor, simplemente use ese valor predeterminado.

```
def printName(firstName, lastName, reverse = False):  
    if reverse:  
        print(lastName+ ', ' + firstName)  
    else:  
        print(firstName, lastName)
```

```
printName('Eric', 'Grimson')  
printName('Eric', 'Grimson', True)
```





EJERCICIOS

Funciones



TIPOS DE DATOS COMPUESTOS

Tuplas, listas, aliasing, mutabilidad y clonación

TUPLAS

- una secuencia ordenada de elementos, puede mezclar tipos de elementos.
- no pueden cambiar valores de elementos, son **inmutables**
- representado con paréntesis

`te = ()` **tupla vacía**

`t = (2, "ucom", 3)`

`t[0]` **⇒ retorna 2**

`t = (2, "ucom", 3) + (5, 6)` **⇒ t retorna (2, "ucom", 3, 5, 6)**

`t[1:2]` **⇒ slice tupla, evalúa a ("ucom",)**

`t[1:3]` **⇒ slice tupla, evalúa a ("ucom", 3)**

`len(t)` **⇒ retorna 3**

`t[1] = 4` **⇒ error, no se puede modificar el objeto**



TUPLAS - USOS

- Conveniente para intercambio de variables

```
x = y  
y = x
```



```
temp = x  
x = y  
y = temp
```



```
(x, y) = (y, x)
```



- Usadas para retornar mas de un valor desde una función

```
def quotient_and_remainder(x, y):
```

```
    q = x // y
```

```
    r = x % y
```

```
    return (q, r)
```

```
(quot, rem) = quotient_and_remainder(4, 5)
```

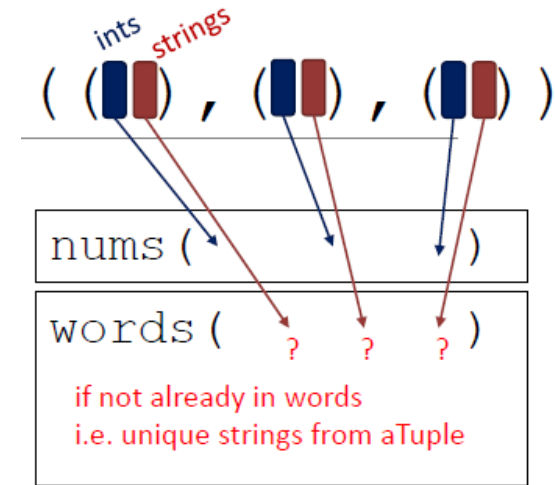
*integer
division*



MANIPULANDO TUPLAS

- Se puede iterar sobre las tuplas

```
def get_data(aTuple):  
    nums= ()  
    words = ()  
    for t in aTuple:  
        nums= nums + (t[0],)  
        if t[1] not in words:  
            words = words + (t[1],)  
    min_n= min(nums)  
    max_n= max(nums)  
    unique_words= len(words)  
    return (min_n, max_n, unique_words)
```



OPERACIONES BÁSICAS

- `len(tupla)` – retorna la longitud total de la tupla
- `del tupla` – borra la tupla
- `cmp(tuple1, tuple2)` – compara elementos de ambas tuplas
- `max(tupla)` – retorna el item de la tupla con el máximo valor
- `min(tupla)` – retorna el item de la tupla con el valor mínimo
- `tuple(seq)` – convierte una lista en tupla



LISTAS

- **secuencia ordenada** de información, accesible por índice.
- una lista se denota por **corchetes**, []
- una lista contiene **elementos**,
 - generalmente homogénea (ej. todos los enteros)
 - puede contener tipos mixtos (no común)
- los elementos de la lista se pueden cambiar para que una lista sea **mutable**.



ÍNDICES Y ORDENACIÓN

`a_list= []` → retorna una lista vacía

`L = [2, 'a', 4, [1,2]]`

`len(L)` → retorna 4

`L[0]` → retorna 2

`L[2]+1` → retorna 5

`L[3]` → retorna `[1,2]`, otra lista!

`L[4]` → da un error

`i= 2`

`L[i-1]` → retorna `'a'` ya que `L[1]='a'`



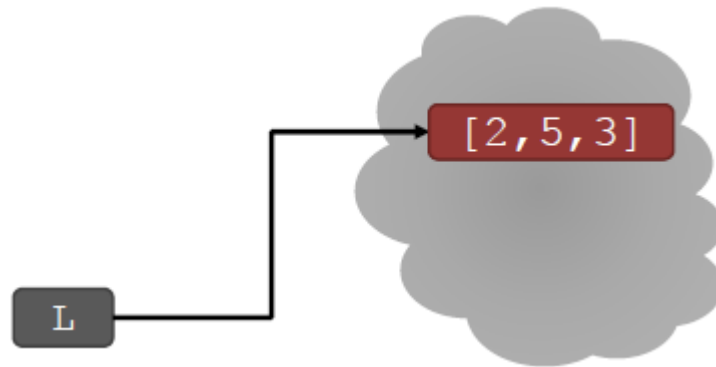
CAMBIANDO ELEMENTOS

- Listas son mutables;
- Asignando un valor a un elemento en un índice, cambia el valor:

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

```
L[1] = 5
```

- L es ahora [2, 5, 3], note que este es el mismo objeto L.



ITERANDO SOBRE LA LISTA

- computar la suma de elementos de una lista
- patrón común, iterar sobre la lista de elementos `L=[2, 5, 3]`

```
total = 0
for i in range(len(L)):
    total += L[i]
print total
```

```
total = 0
for i in L:
    total += i
print total
```

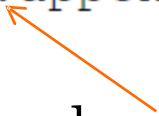
- nota
 - los elementos de la lista son indexados de 0 a `len(L) - 1`
 - `range(n)` va desde 0 a `n-1`



OPERACIONES SOBRE LISTAS - APPEND

- agregar elementos al final de la lista con `L.append(element)`
- muta la lista!

```
L = [2, 1, 3]
L.append(5)    → L es [2, 1, 3, 5]
```



- que significa el punto?
 - listas son objetos Python, todo en Python son objetos;
 - objetos tienen datos;
 - un objeto tiene métodos y funciones;
 - se accede a su información haciendo `nombre_objeto.hacer_algo()`



OPERACIONES SOBRE LISTAS — EXTEND Y +

- Para combinar listas use el operador de concatenación '+'
- Para mutar la lista use `L.extend(alguna_lista)`

```
L1 = [2,1,3]
```

```
L2 = [4,5,6]
```

```
L3 = L1 + L2
```



L3 es [2,1,3,4,5,6]

L1, L2 incambiable

```
L1.extend([0,6])
```



mutado L1 a [2,1,3,0,6]



OPERACIONES CON LISTAS — DEL, POP, REMOVE

- eliminar un elemento en un índice específico con `del (L[índice])`
- eliminar un elemento al final de la lista con `L.pop ()`, devuelve el elemento eliminado
- eliminar un elemento específico con `L.remove (elemento)`
 - busca el elemento y lo elimina.
 - si el elemento aparece varias veces, elimina la primera aparición
 - si el elemento no está en la lista, da un error.

```
L = [2, 1, 3, 6, 3, 7, 0] # do below in order
```

```
L.remove(2) → mutates L = [1, 3, 6, 3, 7, 0]
```

```
L.remove(3) → mutates L = [1, 6, 3, 7, 0]
```

```
del (L[1]) → mutates L = [1, 3, 7, 0]
```

```
L.pop() → returns 0 and mutates L = [1, 3, 7]
```

Todas estas
operaciones
mutan la lista



CONVERTIR LISTAS A STRINGS

- convertir un `string` en una lista con `list(s)`, devuelve una lista donde todos los caracteres de `s` son elementos en `L`;
- puede usar `s.split()`, para dividir un `string` según un carácter específico, se divide en espacios si se llama sin un parámetro;
- usar `' '.join(L)` para convertir una lista de caracteres en un `string`, puede usar un carácter entre comillas para agregar caracteres entre cada elemento

<code>s = "I<3 cs"</code>	→ <code>s</code> is a string
<code>list(s)</code>	→ returns <code>['I', '<', '3', ' ', 'c', 's']</code>
<code>s.split('<')</code>	→ returns <code>['I', '3 cs']</code>
<code>L = ['a', 'b', 'c']</code>	→ <code>L</code> is a list
<code>' '.join(L)</code>	→ returns <code>"abc"</code>
<code>'_'.join(L)</code>	→ returns <code>"a_b_c"</code>



OTRAS OPERACIONES

- `Sort()` y `sorted()`
- `Reverse()`
- y muchos otros. Ver en <https://docs.python.org/3/tutorial/datastructures.html>

```
L=[9,6,0,3]
```

```
sorted(L)      → returns sorted list, does not mutate L
```

```
L.sort()      → mutates L=[0,3,6,9]
```

```
L.reverse()   → mutates L=[9,6,3,0]
```



LISTAS EN MEMORIA

- las listas son **mutables**;
- tiene comportamiento diferente a los tipos inmutables;
- es un objeto en la memoria;
- los nombres de variables apuntan a un objeto;
- cualquier variable que apunte a ese objeto se ve afectada;
- Atención: tener en cuenta al trabajar con listas que pueden ocurrir **efectos secundarios**.



ALIAS

```
1 a = 1
2 b = a
3 print(a)
4 print(b)
5
6 warm = ['red', 'yellow', 'orange']
7 hot = warm
8 hot.append('pink')
9 print(hot)
10 print(warm)
```

alias

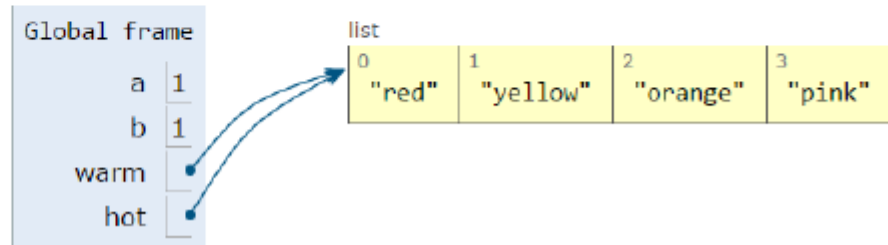
Cambia uno y
cambia el otro

append tiene un
efecto secundario

```
1
1
['red', 'yellow', 'orange', 'pink']
['red', 'yellow', 'orange', 'pink']
```

Frames

Objects



CLONANDO UN LISTA

- Podemos crear una nueva lista y copiar cada elemento usando `chill= cool[:]`

```
1 cool = ['blue', 'green', 'grey']
2 chill = cool[:]
3 chill.append('black')
4 print(chill)
5 print(cool)
```

```
['blue', 'green', 'grey', 'black']
['blue', 'green', 'grey']
```

Frames

Objects

Global frame

cool
chill

list

0	1	2
"blue"	"green"	"grey"

list

0	1	2	3
"blue"	"green"	"grey"	"black"

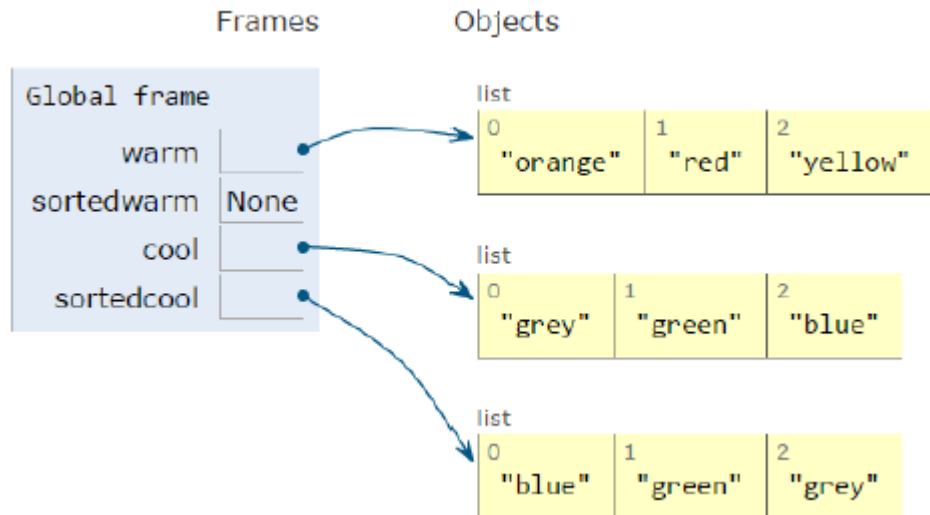


ORDENANDO LISTAS

- `sort()` muta la lista y no retorna nada
- `sorted()` no muta la lista, se debe asignar el resultado a una variable

```
['orange', 'red', 'yellow']  
None  
['grey', 'green', 'blue']  
['blue', 'green', 'grey']
```

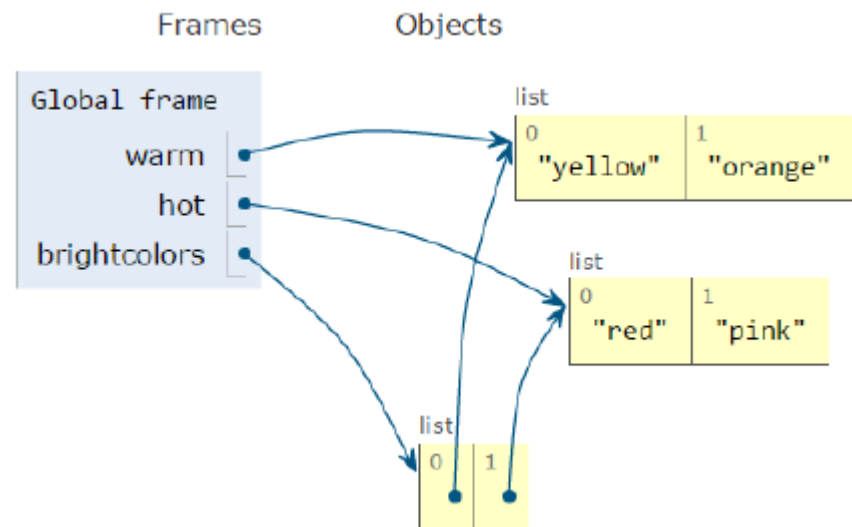
```
1 warm = ['red', 'yellow', 'orange']  
2 sortedwarm = warm.sort()  
3 print(warm)  
4 print(sortedwarm)  
5  
6 cool = ['grey', 'green', 'blue']  
7 sortedcool = sorted(cool)  
8 print(cool)  
9 print(sortedcool)
```



LISTAS ANIDADAS


```
[['yellow', 'orange'], ['red']]  
['red', 'pink']  
[['yellow', 'orange'], ['red', 'pink']]
```


```
1 warm = ['yellow', 'orange']  
2 hot = ['red']  
3 brightcolors = [warm]  
4 brightcolors.append(hot)  
5 print(brightcolors)  
6 hot.append('pink')  
7 print(hot)  
8 print(brightcolors)
```



MUTACIÓN E ITERACIÓN

- Evitar mutar una lista cuando se está iterando sobre ella.

 `def remove_dups(L1, L2):
 for e in L1:
 if e in L2:
 L1.remove(e)`

 `def remove_dups(L1, L2):
 L1_copy = L1[:]
 for e in L1_copy:
 if e in L2:
 L1.remove(e)`

`L1 = [1, 2, 3, 4]
L2 = [1, 2, 5, 6]
remove_dups(L1, L2)`

*Primero clonar la lista,
note que L1_copy = L1,
no es un clon*

- L1 es [2, 3, 4] no [3, 4] Porqué?
 - Python utiliza un contador interno para realizar un seguimiento del índice que está en el loop;
 - la mutación cambia la longitud de la lista pero Python no actualiza el contador;
 - el loop nunca ve el elemento 2.



OPERACIONES BÁSICAS SOBRE LISTAS

- **cmp(list1, list2)** – compara elementos de ambas listas.
- **len(list)** – retorna la longitud total de la lista.
- **max(list)** – retorna el ítem de la lista con valor máximo.
- **min(list)** - retorna el ítem de la lista con valor mínimo.
- **list(seq)** – convierte una tupla en una lista.



MÉTODOS SOBRE LISTAS

- **list.append(obj)** – agrega un elemento **obj** a la lista
- **list.count(obj)** – retorna el conteo de cuántas veces aparece **obj** en la lista
- **list.extend(seq)** – agrega el contenido de **seq** a **list**
- **list.index(obj)** - retorna el índice más bajo en la lista donde aparece **obj**
- **list.insert(index, obj)** – inserta **obj** en **list** en la posición **index**
- **list.pop(obj=list[-1])** - elimina y retorna el último objeto de la lista
- **list.remove(obj)** – elimina **obj** de la **list**
- **list.reverse()** - invierte los objetos de la lista.
- **list.sort([func])** - ordena los objetos de la lista



ALMACENAR INFORMACIÓN RELACIONADA

- como almacenar información sobre los estudiantes?

```
nombres = ['Ana', 'John', 'Denise', 'Katy']  
nota     = ['B', 'A+', 'A', 'A']  
curso    = [2.00, 6.0001, 20.002, 9.01]
```

- una lista separada por cada tema;
- cada lista debe tener la misma longitud;
- la información almacenada en las listas tienen que estar en el mismo índice, cada índice se refiere a información para una persona diferente.



ALMACENAR INFORMACIÓN RELACIONADA

- Como actualizar/recuperar la información?

```
def get_grade(student, name_list, grade_list, course_list):  
    i = name_list.index(student)  
    grade = grade_list[i]  
    course = course_list[i]  
    return (course, grade)
```

- desordenado si tiene un montón de información diferente;
- debe mantener muchas listas y pasarlas como argumentos;
- siempre se debe indexar usando enteros;
- debes recordar cambiar las listas múltiples.



DICCIONARIOS

- bonito para indexar el elemento de interés directamente (no siempre `int`)
- es bonito usar como una estructura de datos, no hay listas separadas

A list

0	Elem 1
1	Elem 2
2	Elem 3
3	Elem 4
...	...

index element

A dictionary

Key 1	Val 1
Key 2	Val 2
Key 3	Val 3
Key 4	Val 4
...	...

custom
index by
label element



UN DICCIONARIO PYTHON

- Almacena pares de datos
 - Clave (key)
 - valor

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

Índice personalizado
para cada elemento

elemento

```
my_dict = {}
```

Diccionario
vacío

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```



key1



val1



key2



val2



key3



val3



key4



val4



BÚSQUEDA EN EL DICCIONARIO

- similar a la indexación en una lista;
- busca la clave;
- devuelve el valor asociado a la clave;
- si no se encuentra la clave, recibe un error.

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

```
grades['John']      → evaluates to 'A+'
```

```
grades['Sylvan']    → gives a KeyError
```



OPERACIONES SOBRE DICCIONARIOS

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- Agregar un elemento

```
grades['Sylvan'] = 'A'
```

- Verificar si la clave existe en el diccionario

```
'John' in grades    → returns True  
'Daniel' in grades → returns False
```

- Borrar una entrada

```
del(grades['Ana'])
```

- Obtener una lista iterable de todas las claves

```
grades.keys() → returns ['Denise', 'Katy', 'John', 'Ana']
```

No
ordenado

- Obtener una lista iterable de todos los valores

```
grades.values() → returns ['A', 'A', 'A+', 'B']
```

No
ordenado



CLAVES Y VALORES

- Valores
 - Cualquier tipo (inmutable y mutable)
 - Pueden ser duplicados
 - Pueden ser listas, o otros diccionarios
- Claves
 - Debe ser único
 - De tipo inmutable (int, float, string, tupla, bool)
 - Claves del tipo float pueden tener un comportamiento inesperado.
- No ordenado

```
d = {4:{1:0}, (1,3):"twelve", 'const':[3.14,2.7,8.44]}
```



OPERACIONES BÁSICAS SOBRE DICCIONARIOS

- **cmp(dict1, dict2)** – compara los elementos de ambos diccionarios.
- **len(dict)** - dá la longitud total del diccionario. Esto sería igual al número de elementos en el diccionario.
- **str(dict)** - produce una representación de cadena de caracteres imprimible de un diccionario.
- **type(variable)** - retorna el tipo de la variable pasada. Si la variable pasada es un diccionario, entonces devolverá un tipo de diccionario.



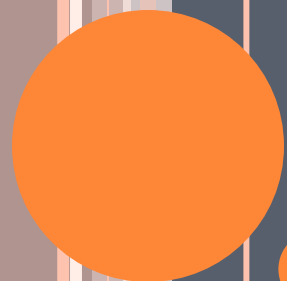
MÉTODOS SOBRE DICCIONARIOS

- **dict.clear()** – elimina todos los elementos del diccionario *dict*
- **dict.copy()** – retorna una copia del diccionario *dict*
- **dict.fromkeys(seq[, value])** – crea un nuevo diccionario con elementos de *seq* como las claves del diccionario.
- **dict.get(key, default=None)** – Para la clave *key*, retorna su valor o el valor por defecto si la clave no está en el diccionario.
- **dict.has_key(key)** - retorna *true* si la clave está en *dict*, *false* de otra forma.
- **dict.items()** – retorna una lista de tuplas (clave, valor).
- **dict.keys()** – retorna una lista de las claves del diccionario *dict*.
- **dict.setdefault(key, default=None)** - similar a *get()*, pero asignará *dict[key]=default* si *key* no está en *dict*.
- **dict.update(dict2)** – substituye los elementos de *dict* por *dict2*'s
- **dict.values()** – retorna una lista de valores del diccionario *dict*.

LIST VS DICT

- secuencia ordenada de elementos;
 - busca elementos por un índice entero;
 - los índices tienen un orden;
 - índices son enteros.
- Mapea “claves” a “valores”;
 - Busca un ítem por otro ítem;
 - El orden no es garantizado;
 - Claves pueden ser del tipo inmutable.





EJERCICIOS

Diccionarios