IA PUCP - Diplomatura de Desarrollo de Aplicaciones de Inteligencia Artificial **Python para Ciencia de Datos**



Introducción (rápida) a Python III

Ver más...



6.0001 Introduction to Computer Science and Programming in Python



View Course

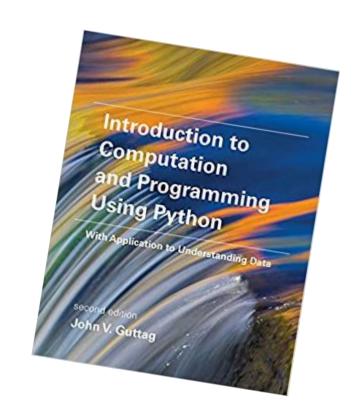
6.0001 is the most common starting point for MIT students with little or no programming experience. This half-semester course introduces computational concepts and basic programming. Students will develop confidence in their ability to apply programming techniques to problems in a broad range of fields. This course uses the Python 3.5 programming language.

Prerequisites: No prior programming experience is necessary to take, understand, or be successful in 6.0001. Familiarity with pre-calculus, especially series, will be helpful for some topics, but is not required to understand the majority of the content.

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/

Ver más...

Guttag, John. *Introduction to Computation and Programming Using Python: With Application to Understanding Data Second Edition.* MIT Press, 2016.
ISBN: 9780262529624.



Se pueden delimitar con comillas simples.

```
cadena = 'Hola Mundo'
```

Se pueden delimitar con comillas dobles.

```
cadena = "Hola Mundo"
```

Se pueden delimitar con comillas simples triples.

```
cadena = '''Hola
Mundo
esta
es una cadena
multilinea'''
```

Se pueden delimitar con comillas dobles triples.

```
cadena = """Hola
Mundo
esta
es una cadena
multilinea"""
```

Longitud de las cadenas

Podemos usar la función len para saber la longitud de una cadena

len("Hola Mundo")

10

Se pueden acceder con el operador []

```
cadena = 'Hola Mundo'
print(cadena[0])
```

En Python se indexa desde 0

Н

Se pueden acceder con el operador []

```
cadena = 'Hola Mundo'
print(cadena[1])
```

 C

Se pueden acceder con el operador []

```
cadena = 'Hola Mundo'
print(cadena[2])
```

Se pueden acceder con el operador []

```
cadena = 'Hola Mundo'
print(cadena[3])
```

a

Se pueden acceder con el operador []

```
cadena = 'Hola Mundo'
print(cadena[-1])
```

Podemos usar índices negativos

 C

Se pueden acceder con el operador []

```
cadena = 'Hola Mundo'
print(cadena[-2])
```

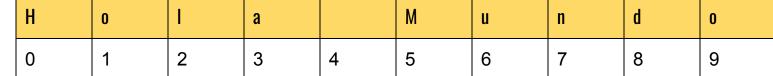
d

Se pueden acceder con el operador []

```
cadena = 'Hola Mundo'
print(cadena[-3])
```

n

Se pueden acceder con el operador []





Se pueden acceder con el operador []

Н	0	I	a		М	u	n	d	0
0	1	2	3	4	5	6	7	8	9
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1



Se pueden realizar slicing con el operador []

```
cadena = 'Hola Mundo'
print(cadena[0:5])
```

Hola

Va desde la posición **0** hasta la posición **5-1**, es decir toma los **5** primeros caracteres

Se pueden realizar slicing con el operador []

```
cadena = 'Hola Mundo'
print(cadena[:5])
```

Hola

Va desde el inicio hasta la posición 5-1. Es equivalente a la operación anterior

Se pueden realizar slicing con el operador []

```
cadena = 'Hola Mundo'
print(cadena[5:])
```

Mundo

Va desde la posición 5 hasta la posición final

Se pueden realizar slicing con el operador []

```
cadena = 'Hola Mundo'
print(cadena[-2:])
```

Toma los dos últimos caracteres

do

Se pueden realizar slicing con el operador []

```
cadena = 'Hola Mundo'
print(cadena[:])
```

Hola Mundo

Toma todos los caracteres

Se pueden realizar slicing con el operador []

```
cadena = 'Hola Mundo'
print(cadena[::2])
```

Toma todos los caracteres, en pasos de 2 en 2.

Hl ud

Н	0	1	a		M	u	n	d	0
0	1	2	3	4	5	6	7	8	9

Se pueden realizar slicing con el operador []

```
cadena = 'Hola Mundo'
print(cadena[1::2])
```

Toma todos los caracteres, en pasos de 2 en 2, comenzando en el 2do (índice 1)

oaMno

Н	0	I	a		М	u	n	d	0
0	1	2	3	4	5	6	7	8	9

Se pueden realizar slicing con el operador []

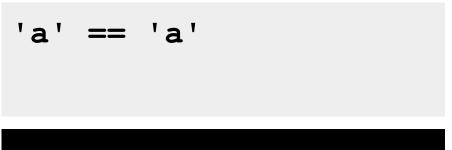
```
cadena = 'Hola Mundo'
print(cadena[::-1])
```

odnuM aloH

Toma todos los caracteres, en pasos de -1, es decir, invierte la cadena

Mayúsculas != minúsculas

En Python (y en muchos otros contextos) una letra minúscula es diferente a una mayúscula

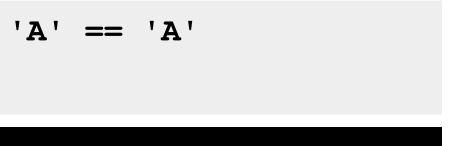




True

Mayúsculas != minúsculas

En Python (y en muchos otros contextos) una letra minúscula es diferente a una mayúscula

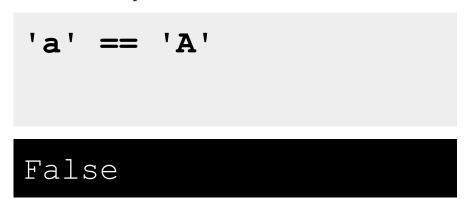




True

Mayúsculas != minúsculas

En Python (y en muchos otros contextos) una letra minúscula es diferente a una mayúscula





Comillas dentro de cadenas delimitadas por comillas

Podríamos acabar generando errores de sintaxis

```
cadena = "Ricardo "El Tigre" Gareca"
print(cadena)
```

```
cadena = "Ricardo "El Tigre" Gareca"
```

SyntaxError: invalid syntax

Las comillas dentro de la cadena, confunden el final de esta.

Comillas dentro de cadenas delimitadas por comillas

Podríamos acabar generando errores de sintaxis

```
cadena = "Ricardo \"El Tigre\" Gareca"
print(cadena)
```

Ricardo "El Tigre" Gareca

Podemos usar el backslash, como un carácter de "escape" indicando que se interprete como contenido de la cadena.

Comillas dentro de cadenas delimitadas por comillas

Podríamos acabar generando errores de sintaxis

```
cadena = 'Ricardo \"El Tigre\" Gareca'
print(cadena)
```

Ricardo "El Tigre" Gareca

O, alternativamente podríamos reemplazar las comillas dobles con simples como delimitadores de la cadena.

Otros caracteres de escape en Python

Código	Resultado
\'	Comilla simple
	Backslash
\n	Nueva línea
\r	Retorno de Carro
\t	Tab
\b	Backspace
\f	Form Feed
\000	Valor octal (base 4)
\xhh	Valor hexadecimal (base 16)

https://www.w3sc hools.com/python /gloss_python_es cape_characters. asp

¿Retorno del carro?



¿Identifica el **retorno del carro**? Ver a partir del 01:53 minutos.

Die Brandenburger Symphoniker mit "The Typewriter" in Brandenburg/Germany 2012

Dando formato a las cadenas: en el orden de aparición

Podemos dar formato a las cadenas con el método .format()

```
cad = "{} {}".format("Hola", "Mundo")
print(cad)
```

Hola Mundo

Dando formato a las cadenas: en el orden indicado

Podemos dar formato a las cadenas con el método .format()

```
cad = "{1} {0}".format("Hola", "Mundo")
print(cad)
```

Mundo Hola

Dando formato a las cadenas: con keywords

Podemos dar formato a las cadenas con el método .format()

```
cad = "Hey {n}".format(n="Juan")
print(cad)
```

Hey Juan

Dando formato a las cadenas: números de punto flotante

Podemos dar formato a las cadenas con el método .format()

```
cad = "pi = {0:.2f}".format(3.14159)
print(cad)
```

```
pi = 3.14
```

Podemos usar el formato para redondear números decimales

Dando formato a las cadenas: números de punto flotante

Podemos dar formato a las cadenas con el método .format()

```
cad = "pi = {0:.4f}".format(3.14159)
print(cad)
```

$$pi = 3.1416$$

Podemos aumentar la cantidad de posiciones decimales a imprimir

Operaciones "aritméticas" con cadenas

Podemos concatenar dos cadenas con el operador +

```
print("Hola" + "Hola")
```

HolaHola

Operaciones "aritméticas" con cadenas

Podemos repetir una cadena con el operador *

print("Hola"*3)

HolaHolaHola

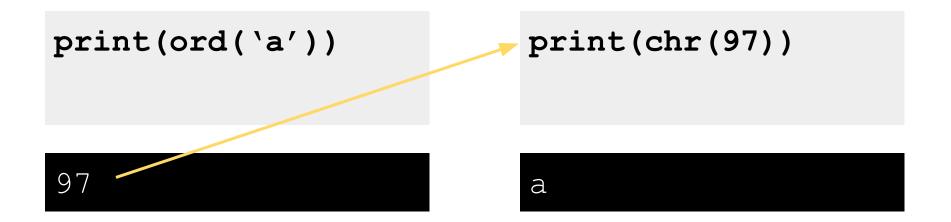
Código de los caracteres

Los caracteres tienen una representación numérica proveniente de una tabla como ASCII.

```
print(ord('a'))
```

Código de los caracteres

Los caracteres tienen una representación numérica proveniente de una tabla como ASCII.



Otros métodos de cadenas: .isdigit()

Podemos verificar si todos los caracteres en una cadena son dígitos

```
cad = "1234"
print(cad.isdigit())
```

True

Otros métodos de cadenas: .isdigit()

Podemos verificar si todos los caracteres en una cadena son dígitos

```
cad = "3.1416"
print(cad.isdigit())
```

False

Otros métodos de cadenas: .isalpha()

Podemos verificar si todos los caracteres en una cadena son alfabéticos

```
cad = "HolaMundo"
print(cad.isalpha())
```

True

Otros métodos de cadenas: .isalpha()

Podemos verificar si todos los caracteres en una cadena son alfabéticos

```
cad = "Hola Mundo"
print(cad.isalpha())
```

False

Otros métodos de cadenas: .endswith()

Podemos verificar si una cadena termina con una subcadena en particular

```
cad = "Hola Mundo"
print(cad.endswith("Mundo"))
```

True

Otros métodos de cadenas: .index()

Podemos ubicar la primera ocurrencia de una subcadena dentro de otra

```
cad = "Hola Mundo"
print(cad.index("Mundo"))
```

5

Si la cadena buscada no estuviera, dará error

Otros métodos de cadenas: .find()

Podemos ubicar la primera ocurrencia de una subcadena dentro de otra

```
cad = "Hola Mundo"
print(cad.find("Mundo"))
```

5

Similar a la función index pero si la cadena buscada no estuviera, retornará -1

Otros métodos de cadenas: .replace(old, new)

Podemos buscar y reemplazar ocurrencias de una subcadena dentro de otra

```
cad = "Estimado [usuario],..."
print(cad.replace("[usuario]", "Juan"))
```

Estimado Juan, ...

Otros métodos de cadenas: .count ()

Podemos contar cuántas veces ocurre una subcadena dentro de otra

```
cad = "Hola Mundo"
print(cad.count("o"))
```

Otros métodos de cadenas: in

Podemos verificar si una cadena se encuentra dentro de otra

```
cad = "Hola Mundo"
print("Mundo" in cad)
```

True

Otros métodos de cadenas: .upper()

Podemos convertir una cadena a mayúsculas

```
cad = "Hola Mundo"
print(cad.upper())
```

HOLA MUNDO

Otros métodos de cadenas: .lower()

Podemos convertir una cadena a minúsculas

```
cad = "Hola Mundo"
print(cad.lower())
```

hola mundo

Listas

Listas en Python

Las listas en Python se declaran con corchetes []. A continuación crearemos una lista con los números del 1 al 5.

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

Listas en Python

También podemos crear una lista vacía

```
lista_vacia = []
```

Listas en Python

Las listas son colecciones con orden, y pueden albergar más de un tipo de dato

```
lista = ["Hola", 2, 3.3, True, None]
```

Longitud de las listas

Podemos usar la función len para saber la longitud de una lista

```
len([1,2,3,4,5])
```

口)

Se pueden acceder con el operador []

```
lista = [100,101,102]
print(lista[0])
```

En Python se indexa desde 0

Se pueden acceder con el operador []

```
lista = [100,101,102]
print(lista[1])
```

Se pueden acceder con el operador []

```
lista = [100,101,102]
print(lista[2])
```

Accediendo a elementos dentro de una lista para modificarlos

Se pueden acceder y modificar con el operador []

```
lista = [100,101,102]
lista[1] = 999
print(lista)
```

```
[100, 999, 102]
```

Se pueden acceder con el operador []

```
lista = [100,101,102]
print(lista[-1])
```

Podemos usar índices negativos

Se pueden acceder con el operador []

```
lista = [100,101,102]
print(lista[-2])
```

Se pueden acceder con el operador []

```
lista = [100,101,102]
print(lista[-3])
```

Se pueden realizar slicing con el operador []

```
lista = [100,101,102,103,104,105,106]
print(lista[0:5])
```

[100, 101, 102, 103, 104]

Va desde la posición **0** hasta la posición **5-1**, es decir toma los **5** primeros valores

Se pueden realizar slicing con el operador []

```
lista = [100,101,102,103,104,105,106]
print(lista[:5])
```

[100, 101, 102, 103, 104]

Va desde el inicio hasta la posición 5-1. Es equivalente a la operación anterior

Se pueden realizar slicing con el operador []

```
lista = [100,101,102,103,104,105,106]
print(lista[5:])
```

[105, 106]

Va desde la posición 5 hasta la posición final

Se pueden realizar slicing con el operador []

```
lista = [100,101,102,103,104,105,106]
print(lista[-3:])
```

[104, 105, 106]

Toma los tres últimos caracteres

Se pueden realizar slicing con el operador []

```
lista = [100,101,102,103,104,105,106]
print(lista[::2])
```

[100, 102, 104, 106]

Toma todos los elementos, en pasos de 2 en 2.

100	101	102	103	104	105	106
0	1	2	3	4	5	6

Se pueden realizar slicing con el operador []

```
lista = [100,101,102,103,104,105,106]
print(lista[1::2])
```

[101, 103, 105]

Toma todos los elementos, en pasos de 2 en 2, comenzando en el 2do (índice 1)

100	101	102	103	104	105	106
0	1	2	3	4	5	6

Podemos tomar "porciones" de las listas

Se pueden realizar slicing con el operador []

```
lista = [100,101,102,103,104,105,106]
print(lista[::3])
```

[100, 103, 106]

Toma todos los elementos, en pasos de 3 en 3.

100	101	102	103	104	105	106
0	1	2	3	4	5	6

Podemos tomar "porciones" de las listas

Se pueden realizar slicing con el operador []

```
lista = [100,101,102,103,104,105,106]
print(lista[::-1])
```

[106, 105, 104, 103, 102, 101, 100]

Toma todos los caracteres, en pasos de -1, es decir, invierte la lista

Operaciones "aritméticas" con listas

Podemos concatenar dos listas con el operador +

[1,2,3,4,5]

Operaciones "aritméticas" con listas

Podemos repetir una lista con el operador *

```
print([1,2,3,4]*3)
```

```
[1,2,3,4,1,2,3,4,1,2,3,4]
```

Podemos agregar un elemento a una lista

Se pueden realizar con el método .append ()

```
lista = [100,101,102]
lista.append(103)
print(lista)
```

```
[100, 101, 102, 103]
```

.append agrega el nuevo valor al final

Podemos agregar múltiples elementos a una lista

Se pueden realizar con el método .extend()

```
lista = [100,101,102]
lista.extend(range(5))
print(lista)
```

[100, 101, 102, 0, 1, 2, 3, 4]

.extend agrega los nuevos valores al final

Recorrer una lista con for

Se puede recorrer una lista con for en función de la longitud de la lista

```
lista = [100,101,102]
for i in range(len(lista)):
   print(lista[i])
```

```
100
101
102
usamos i para indexar la lista y acceder a los elementos
```

Recorrer una lista con for



Se puede recorrer una lista con for

```
lista = [100,101,102]
for elem in lista:
   print(elem)
```

100101102

elem tomará un valor diferente de la lista en cada iteración

Ordenar una lista (de menor a mayor)

Podemos ordenar una lista

```
a = [1,4,3,2]
a.sort()
print(a)
```

```
[1,2,3,4]
```

Ordenar una lista (de mayor a menor)

Podemos ordenar una lista

```
a = [1,4,3,2]
a.sort(reverse=True)
print(a)
```

```
[4,3,2,1]
```

Ordenar una lista (de mayor a menor)

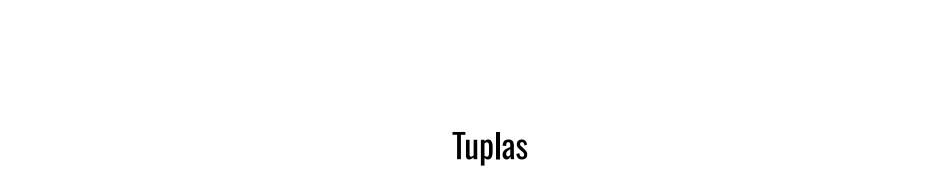


Podemos ordenar una lista

```
a = [1,4,3,2]
a.sort()
print(a[::-1])
```

[4,3,2,1]

En esta "opción" ordenamos la lista a y luego la invertimos



Declarar tuplas

Podemos declarar una tupla con paréntesis

```
a = (1,2,3)
print(type(a))
```

tuple

Declarar tuplas

Podemos declarar una tupla sin paréntesis

```
a = 1,2,3
print(type(a))
```

tuple

Accediendo a elementos dentro de una tupla

Se pueden acceder con el operador []

```
tupla = (100,101,102)
print(tupla[0])
```

En Python se indexa desde 0

100

La naturaleza inmutable de una tupla

No se pueden cambiar los valores de una tupla (es inmutable)

```
tupla = (100,101,102)
tupla[1]=999
```

```
----> 2 tupla[1]=999

TypeError: 'tuple' object does not support item assignment
```

Declarar tuplas

Podemos declarar una tupla sin paréntesis

```
def f(x):
    return x**2, x**3
print(type(f(3)))
```

Esto nos permite devolver múltiples valores como resultado de una función

tuple

Desempaquetar tuplas

Esto nos permite realizar una "doble asignación" en una línea de código

```
a, b = 2, 3
print(a)
print(b)
```

2

3

Desempaquetar tuplas

...e intercambiar valores

```
a=1
b=2
a, b = b, a
print(a)
print(b)
```



Declarar un diccionario

Podemos declarar un diccionario vacío

```
d = {}
print(type(d))
```

dict

Declarar un diccionario

Podemos declarar un diccionario con pares llave: valor (key: value)

```
aeropuerto = {"LIM":"Lima", "MIA":"Miami"}
print(type(aeropuerto))
```

dict

Acceder a un elemento

Podemos acceder a un elemento de un diccionario usando su llave

```
aeropuerto = {"LIM":"Lima", "MIA":"Miami"}
print(aeropuerto["LIM"])
```

Lima

Agregar un elemento

Podemos agregar un elemento de un diccionario usando la llave. Si la llave ya existiera, modificará su valor.

```
aeropuerto = {"LIM":"Lima", "MIA":"Miami"}
aeropuerto["GRU"]="Guarulhos"
print(aeropuerto["GRU"])
```

Guarulhos



Declarar un conjunto

Podemos declarar un conjunto vacío

```
c1 = set()
print(type(c1))
```

set

Declarar un conjunto

Podemos declarar un conjunto con valores

```
c1 = {1,2,3,4}
print(type(c1))
```

set

Operaciones de conjuntos: Unión

Podemos declarar un conjunto con valores

```
c1 = {1,2,3,4}
c2 = {2,4,5}
print(c1|c2)
```

```
{1,2,3,4,5}
```

Operaciones de conjuntos: Intersección

Podemos declarar un conjunto con valores

```
c1 = {1,2,3,4}
c2 = {2,4,5}
print(c1&c2)
```

{2,4}

Operaciones de conjuntos: Diferencia

Podemos declarar un conjunto con valores

```
c1 = {1,2,3,4}
c2 = {2,4,5}
print(c1-c2)
```

{1,3}

Operaciones de conjuntos: Diferencia

Podemos declarar un conjunto con valores

```
c1 = {1,2,3,4}
c2 = {2,4,5}
print(c2-c1)
```

{ 5 }

La operación no es simétrica

Operaciones de conjuntos: Pertenencia

Podemos declarar un conjunto con valores

True

Funciona con listas, pero en conjuntos la operación está optimizada

Operaciones de conjuntos: No-Pertenencia

Podemos declarar un conjunto con valores

```
c1 = {1,2,3,4}
print(1 not in c1)
```

False