Práctica 0: Introducción a Python

Gonzalo Martínez, Irene Rodríguez, Mario Mingo

¿Qué es python?

- Es de propósito general
- Interpretado
- Tipado dinámico
- Lenguaje multiparadigma: procedural, funcional, orientado a objetos
- Fácil de utilizar
- Fácil de aprender

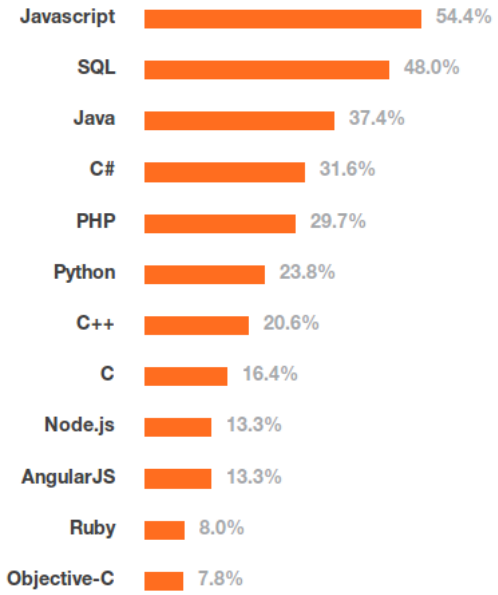
¿Qué es python?

- Creador: Guido van Rossum
- Primera versión pública 1991
- Dos versiones en paralelo:
  - Versión 2.x (sin soporte desde 1 de enero de 2020)
  - Versión 3.x a partir de 2008
- El nombre viene de Monty Python Flying Circus
- Esta implementado en Ansi C
- Es de código abierto
- Es multiplataforma: Linux, Windows, OSX…

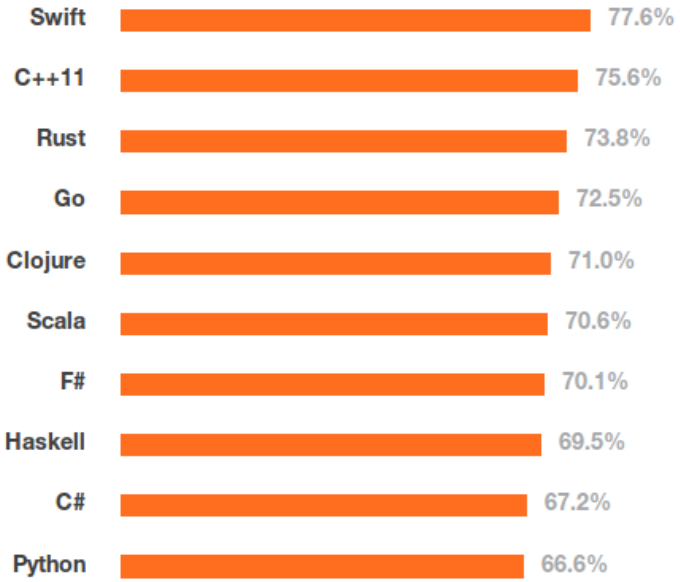
Stackoverflow survey 2015

Tecnologías

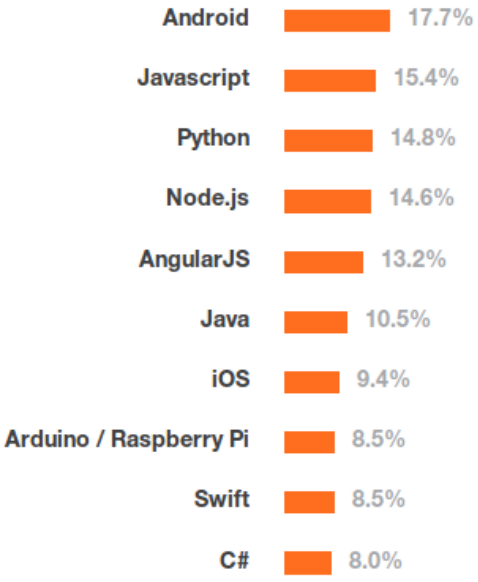
Más populares



Más queridas



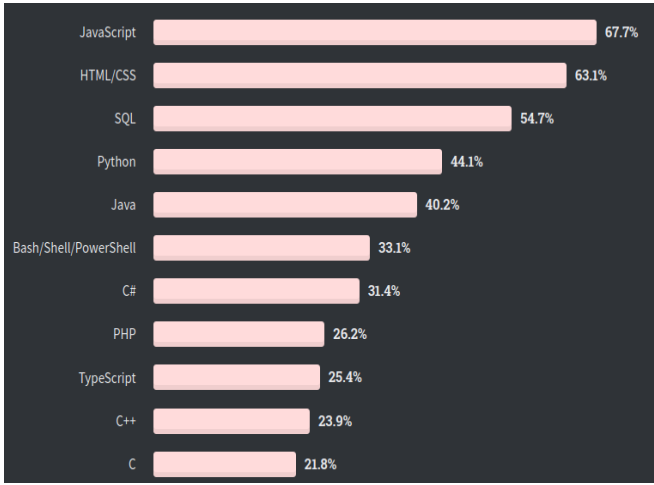
Más deseadas



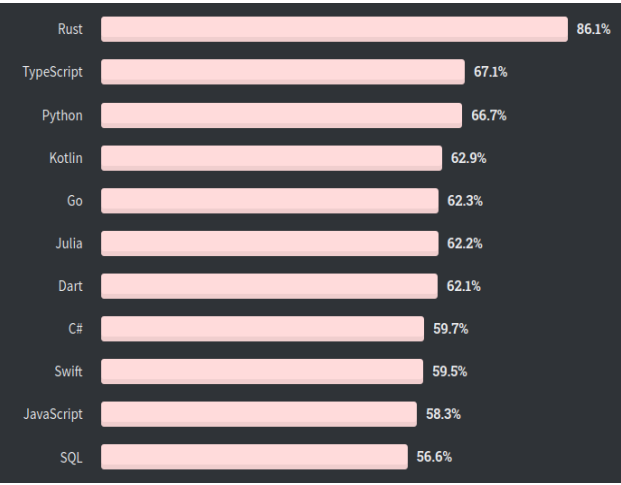
Stackoverflow survey 2020

Tecnologías

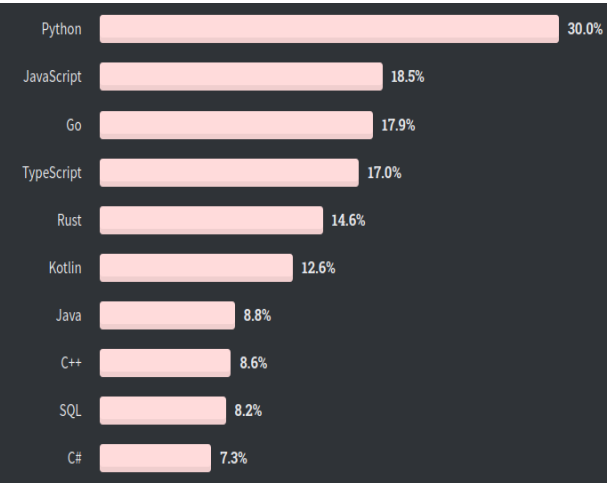
Más populares



Más queridas



Más deseadas



Configuración python

- Herramientas necesarias:
  - **Interprete python**: usaremos interprete **ipython** y **notebooks**
    - **Ipython**: shell interactivo que añade funcionalidades extra al modo interactivo incluido con Python (resaltado de líneas y errores mediante colores, una sintaxis adicional para el Shell, autocompletado, etc.).
    - **Ipython notebooks**: ahora denominados **Jupyter Notebooks**. Son un entorno interactivo de programación en el que se puede combinar texto, código, ecuaciones, figuras, etc. Para ello se compone de celdas que pueden ser de tipo código o tipo texto (acepta lenguaje *markdown*).
  - **Editor de texto**

Configuración python

- En el laboratorio:
  - **Anaconda:** distribución totalmente gratuita de python que incluye más de 300 paquetes de Python (numpy, scikit, matplotlib, etc.)
  - En las prácticas trabajaremos preferentemente con las versiones 3.6 o 3.7

IMPORTANTE! Para poder usar Anaconda en los laboratorios de prácticas, debe establecerse la variable de entorno `PATH` como:

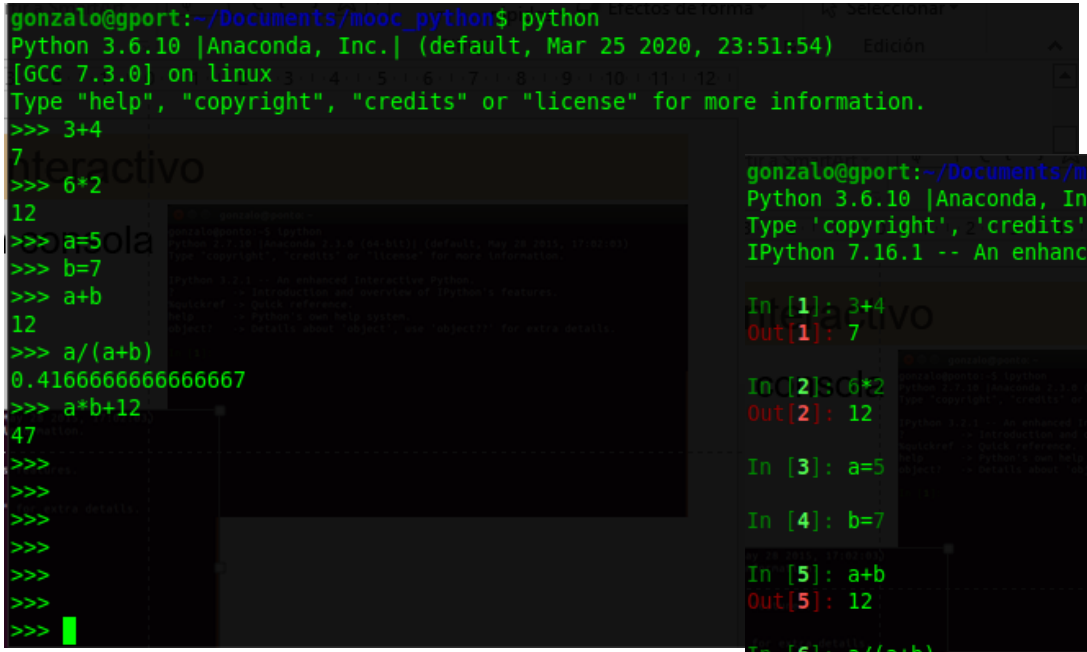
```
# export PATH=${PATH}:/opt/anaconda-3.7.0/bin
```

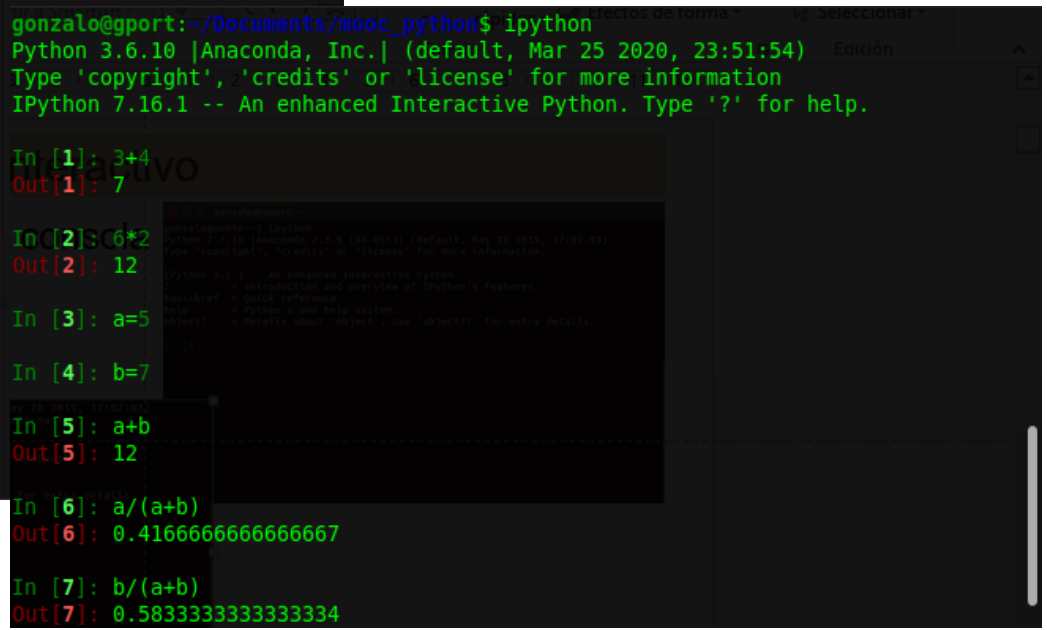
Para arrancar el servidor de notebooks, se debe ejecutar desde el directorio de instalacion (`/opt/anaconda-3.7.0/`) de la siguiente forma:

```
# cd /opt/anaconda-3.7.0
# ./bin/ipython notebook $HOME
```

Ipython: Usando el modo interactivo

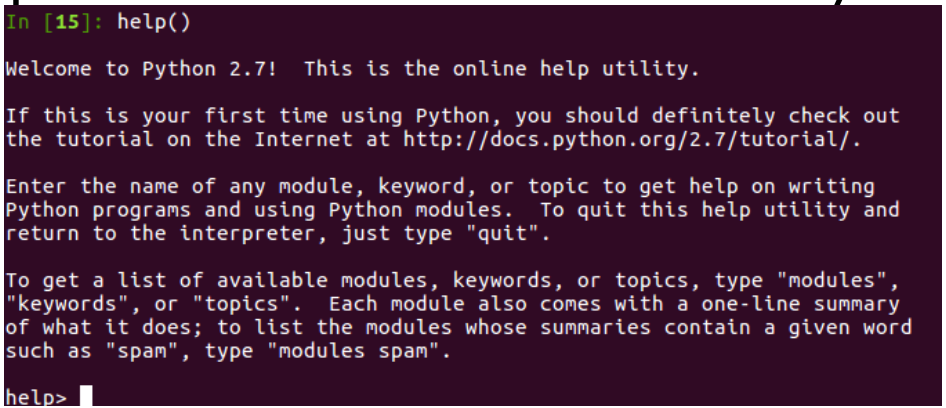
- Ejecutar **ipython** en consola



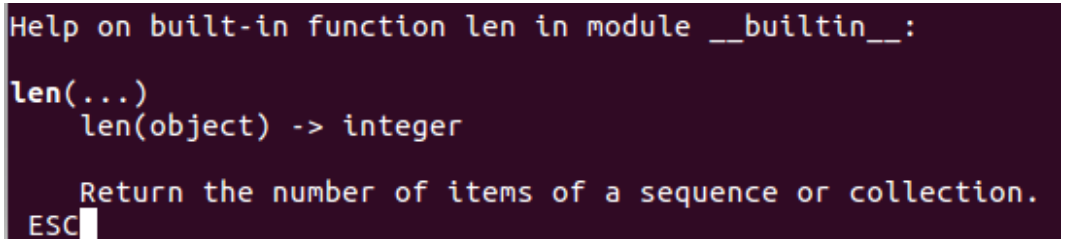


Ipython: Usando el modo interactivo

- Función **`help()`**: Sin parámetros entra en modo de ayuda interactiva



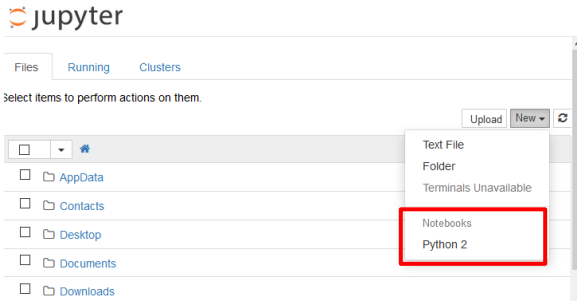
- También se puede llamar con el nombre de la función de la que se necesita ayuda. Por ejemplo: **`help(len)`**

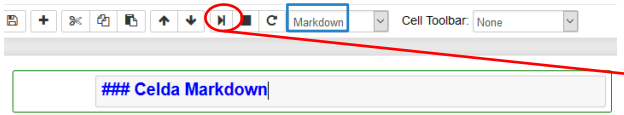


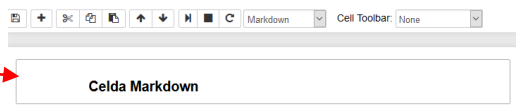
Jupyter Notebooks

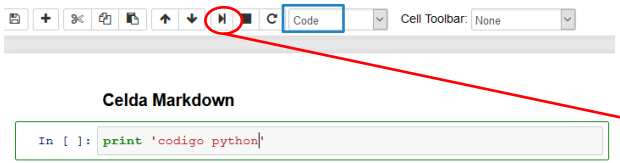
- Ejecutar **Jupyter notebook** en consola:

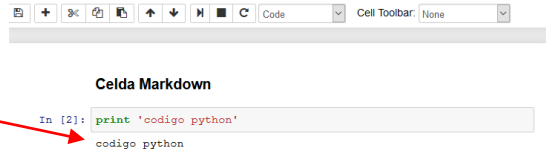
```
irene@PORTEGE-Z30-B:~$ ipython notebook
[TerminalIPythonApp] WARNING | Subcommand `ipython notebook` is deprecated and will be removed in future versions.
[TerminalIPythonApp] WARNING | You likely want to use `jupyter notebook`... continue in 5 sec. Press Ctrl-C to quit now.
[I 09:37:51.562 NotebookApp] Serving notebooks from local directory: /home/irene
[I 09:37:51.562 NotebookApp] 0 active kernels
[I 09:37:51.562 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888/
[I 09:37:51.562 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
```











Algunas referencias

- "An introduction to python" Guido van Rossum
  - En la biblioteca
  - Y online: https://docs.python.org/3.7/tutorial/
- "Learning Python" Lutz, Mark
  - Disponible en la biblioteca en formato electrónico.
- Referencia del lenguaje en:

  https://docs.python.org/3.7/reference/index.html
- Tutoriales a diferentes niveles y para diferentes usos:

  https://pythonspot.com/

Filosofía de python

- Python busca:
  - simplicidad
  - elegancia
  - legibilidad
- Zen of Python
  - Beautiful is better than ugly.
  - Explicit is better than implicit.
  - Simple is better than complex.
  - Complex is better than complicated.
  - Flat is better than nested.
  - …teclea `import this` en la consola para la lista completa

Filosofía de python (Definiciones)

- Pitónico (pythonic):

  Código python que sigue adecuadamente la filosofía de python, que utiliza correctamente sus construcciones, que es legible, minimalista…

- Pitonisa/o:

  Buenos programadores de python

El primer código de python

parrot.py

```
#!/usr/bin/python
# Importamos el modulo del sistema
import sys

def repeat(text):
    """Imprime la cadena text pasada por argumento
       no hace ninguna comprobación sobre el tipo de text  """
    print (text)

if __name__ == '__main__':
    repeat(sys.argv[1])
```

Permite ejecución directa en linux

Comentarios de línea

Importado de módulos

Definición de función

Código del main

- Ejecución desde línea de comandos: python+fichero.py

```
gonzalo@ponto:$ python parrot.py hola
hola
gonzalo@ponto:$ ./parrot.py adios
adios
gonzalo@ponto:$
```

El primer código de Python… en un Notebook

- **Ejercicio**:
  - Abrir un Jupyter notebook que contenga una celda de texto (Markdown) y una celda de código.
  - Definir en la celda de código la función `repeat` del ejemplo anterior y llamarla con una cadena cualquiera como parámetro.
  - Ejecutar la celda.
  - Guardar el notebook en formato ipython notebook (ipynb): `File` → `rename`
  - Exportar el notebook como html: `File` → `Download as`

Variables y tipos de datos

- En python todos los elementos son objetos incluyendo los tipos básicos como **`int`**.
- No hay declaración de variables.
- Las variables son referencias a objetos
- Se hacen comprobaciones en ejecución de los tipos de las variables
- Los tipos básicos son:
  - Booleano
  - Numéricos: int, float, complex.
  - Secuencias: cadenas, listas, tuplas.
  - Otros: diccionarios, conjuntos, etc.

Tipos numéricos: int, long, float, complex

- **`int`**: tipo entero implementado de precisión ilimitada
- **`float`**: tipo de coma flotante implementado usando **`double`** de C
- **`complex`**: tipo complejo. Dado un complejo z se accede a la parte real e imaginaria con z.real y z.imag
- Al trabajar con números se crea el tipo mínimo que puede representar al número. Esto es:

  int < float < complex

```
In [30]: a = 4        # Crea un int
In [31]: a = 3.14     # Crea un float
In [32]: z = 3 + 4j   # Crea un complex
```

Operadores numéricos básicos

- Funcionamiento similar a C y otros lenguajes
- Si pueden mezclar distintos tipos numéricos en las operaciones.
- Python hace la operación usando el tipo más complejo de forma automática. La complejidad viene dada por:

  int < float < complex.

- Estos operadores se pueden combinar con la asignación: +=, -=, *=, etc.

Operador	Descripción
`a + b`	Suma de a y b
`a - b`	Resta
`a * b`	Producto
`a / b`	División *
`a // b`	División entera
`a % b`	Resto
`a**b`	Potencia a^b

*En python 2.X la división será entera si a y b son enteros. Si no, es división con decimales.
En python 3.X siempre es división con decimales

Operadores numéricos básicos (ejemplos)

```
In [36]: a=4

In [37]: b=5

In [38]: a+b
Out[38]: 9

In [39]: a-b
Out[39]: -1

In [40]: a*b
Out[40]: 20

In [41]: a/b
Out[41]: 0.8

In [42]: a//b
Out[42]: 0

In [43]: a%b
Out[43]: 4

In [44]: a**b
Out[44]: 1024

In [45]: (1j)**2
Out[45]: (-1+0j)

In [46]:
```

- Algunos comentarios:
  - Se puede asignar n valores a n variables en una línea usando comas.
  - Se puede también hacer:

```
a, b = b, a  # Intercambio
a = b = c = 7 # Asignación múltiple
```

  - Las variables en python son referencias a objetos. Realmente no hay asignación, simplemente se liga una variable a un objeto.

Cadenas

- Se representan con comillas simples o dobles:

```
c1 = 'Soy una cadena'
c2 = "También lo soy 'yo'"    # Se pueden usar comillas dentro de la cadena
c3 = 'y "yo"'                 #   del tipo contrario
```

- Longitud de una cadena:

```
>>> len("hola")
4
```

- Operadores para cadenas:
  - Concatenar con +:

```
>>> "Hola " + "Mundo"
'Hola Mundo'
```

  - Replicar con *:

```
>>> "H"*5
'HHHHH'
```

- Son **inmutables** ¡¡Cada operación crea una nueva cadena!!

Más sobre cadenas

- Se pueden incluir caracteres especiales como en C: \n, \t, etc.
- Creación de cadenas de varias líneas con triples comillas:

```
>>> c1 = """Soy una cadena
en varias líneasssssssss"""
```

- Las cadenas con triples comillas también se usan como comentarios.
- En python las cadenas se representan internamente con unicode por lo que podemos escribir directamente caracteres especiales en las cadenas o introducir caracteres Unicode con \u:

```
>>> s = "mi \u00f1u es muy majo"
>>> s
'mi ñu es muy majo'
>>> len(s)
10
```

Indexación en cadenas (y secuencias)

- El operador [ ] se usa para acceder a los caracteres de las cadenas (y a los elementos de las secuencias como veremos más adelante).

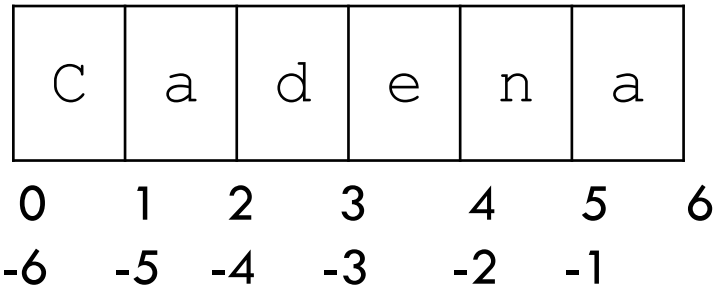
```
>>> c = "cadena"
>>> c[0]    # Se indexa empezando en 0
'c'
>>> c[-2]   # Se puede indexar desde el final con números negativos
'n'


>>> c[1:4]  # Se pueden obtener porciones de la cadena usando dos índices
'ade'

>>> c[6]    # Si te pasas del tamaño da error
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range

>>> c[4] = 'r'   # No se pueden asignar caracteres, las cadenas son inmutables
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Indexación en cadenas (y secuencias)

- Hay tres formas de obtener subsecuencias de la cadena:

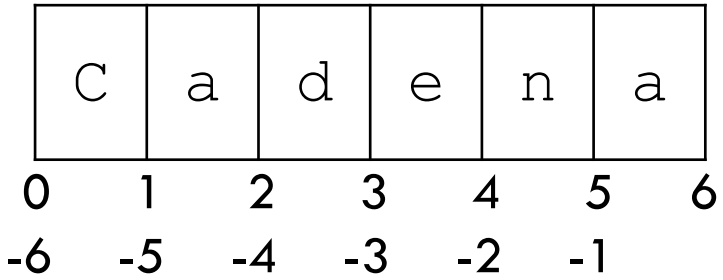
```
c[posicion]
c[inicio:fin]
c[inicio:fin:paso]
```

- Selecciona de `inicio` a `fin-1` de `paso` en `paso`.
- Valores por omisión: `inicio=0`, `fin=len(c)` y `paso=1`
- Una forma de verlo es pensar que los índices están entre los caracteres:



- **Cada operación crea una cadena nueva**

Indexación en cadenas (y secuencias)




```
>>> c = "cadena"
>>> c[2:len(c)]
'dena'
>>> c[2:] # Mejor usando valores por omisión
'dena'
>>> c[0:6:2] # Un carácter de cada dos
'cdn'
>>> c[::2]  # Mejor usando valores por omisión
'cdn'

>>> c[-3:]  # Los últimos tres elementos

>>> c[::-1]    # Con pasos negativos recorremos la lista al revés
'anedac'
```

Algunos métodos de cadenas I

Método	Descripción
`c.lower(), c.upper()`	Convierte a minúsculas, mayúsculas
`c.strip([chars])`	Elimina blancos (o `chars` si se especifican) de los extremos de la cadena
`c.split([sep])`	Devuelve una lista de cadenas partiendo la cadena por los espacios (o `sep` si indicado)
`c.join(iterable)`	Une los elementos del `iterable` en una única cadena separada por el contenido de c. Es el complementario de split
`c.find(sub[,ini[,fin]])`	Busca la cadena `sub` en c y devuelve el índice. Si no lo encuentra devuelve -1. Se puede especificar límites de búsqueda

Algunos métodos de cadenas II

Método	Descripción
`c.replace(old, new[, count])`	Reemplaza `old` por new hasta `count` veces
`c.center(width[, fillchar])`	Centra la cadena en otra de longitud `width` rellenando con `fillchar` o blancos
`c.ljust(width[, fillchar])`	Justifica a la izquierda la cadena en otra de longitud `width` rellenando con `fillchar` o blancos
`c.rjust(width[, fillchar])`	Justifica a la derecha la cadena en otra de longitud `width` rellenando con `fillchar` o blancos

Más métodos en: https://docs.python.org/3.7/library/stdtypes.html#string-methods

Ejemplos

```
>>> c = "cadena"
>>> c.upper()
'CADENA'
>>> ' Hola   '.strip()
'Hola'
>>> 'a,b,,d'.split(',')
['a', 'b', '', 'd']
>>> '-'.join(['a', 'b', '', 'd'])
'a-b--d'

>>> 'hola'.replace('ho', 'Oh ').replace('la', 'la la')
'Oh la la'
>>> 'hola'.center(12, '*')
'****hola****'
```

Ejercicios I

```
# Dada una cadena pon los caracteres con índice par al principio y con índice
# impar después separador por un guion. Ejemplo "ABCDEF" debe dar "ACE-BDF"
c = "ABCDEF"
c =

# Primera letra a mayúscula y resto en minúsculas
c="hOlA"
c =

# Cambiar espacios por _ y poner en minúsculas la extensión
# Suponer que la extensión tiene 3 caracteres
fichero = 'Foto de vacaciones.JPG'
fichero =
```

Ejercicios II

```
# Combina join y split para que devuelva exactamente la misma cadena
# original
sep = "-"
c = "A-B-C-D-E-F"
c =

# rodea la cadena c con > por la izquierda y < por la derecha hasta
#  un total de 80 caracteres
c = "recuadrar"
c =
```

Función format

- Funciona de forma similar al operador % pero es más potente.
- Se especifica en general con `{[modificadores]no. de parámetro}` dentro de la cadena y luego se especifican valores usando format:

```
>>> "{0}+{0} es {1}".format(2,4)
'2+2 es 4'
```

- Se pueden usar los valores varias veces
- Algunos modificadores para definir longitud, completar con ceros, etc

```
>>> "2.0 + {1:05.2f} es {2:f}".format(2, 3.14159, 2+3.14159)
'2.0 + 03.14 es 5.141590'
```

Más en: https://docs.python.org/3.7/library/string.html#formatstrings

Función `print()` - I

```
print(*objects, sep=' ', end='\n', file=sys.stdout)
```

- La función `print()` recibe una lista de objetos y los imprime incluyendo un espacio entre cada objeto y un salto de línea al final:

```
>>> print("La distancia es de", 2+2, "Km.")
La distancia es de 4 Km.
```

- Se puede especificar separador y elemento final

```
>>> print("La distancia es de", 2+2, "Km.", sep="-", end="!")
La distancia es de-4-Km.!
```

Listas I

- Tipo compuesto muy versátil.
- Se representan con corchetes [ ] y los elementos separados por comas.
- Permite incluir elementos de distintos tipos

```
>>> a = [3, 'pi', 3.14, True]
[3, 'pi', 3.14, True]
```

- Es una secuencia como las cadenas y `len` te da su longitud

```
>>> len(a)
4
```

Listas II

- Como con las cadenas, se accede a sus elementos con el operador ":"

```
>>> a = [3, 'pi', 3.14, True]
>>> a[2]
3.14
>>> a[0:1]
[3, 'pi']
>>> a[-1]
True
>>> a[0:-1]
[3, 'pi', 3.14]
>>> a[::2]
[3, 3.14]
```

Listas III

- Como con las cadenas, el operador + concatena

```
>>> a + ['e']
[3, 'pi', 3.14, True, 'e']
```

- Como con las cadenas, el operador * replica

```
>>> 2*a
 [3, 'pi', 3.14, True, 3, 'pi', 3.14, True]
```

Listas IV

- Las listas son **mutables**. Podemos hacer las siguientes ediciones:
  - Asignar elementos:

```
>>> a[1] = 'rodaballos'
>>> a[0] += 3
>>> a
[6, 'rodaballos', 3.14, True]
```

  - Reemplazar varios elementos:

```
>>> a[0:4:2] = [4, 2]
>>> a
[4, 'rodaballos', 2, True]
```

Listas V

- Más ediciones en listas:
  - Eliminar varios elementos

```
>>> a[2:] = []
>>> a
[4, 'rodaballos']
```

  - Insertar elementos:

```
>>> a = [3]
>>> a[0:0] = [1]
>>> a[1:1] = [2]
>>> a[3:3] = [2,1]
[1, 2, 3, 2, 1]
```

Listas VI

- También se pueden anidar listas:

```
>>> a = [2, 3]
>>> b = [1, a, 4]
>>> b
 [1, [2, 3], 4]
>>> len(b)
3
```

Algunos métodos de listas I

Método	Descripción
`a.append(x)`	Añade elemento x al final de la lista
`a.extend(L)`	Añade los elementos de la secuencia L al final de la lista
`a.insert(i,x)`	Añade elemento x en la posición i de la lista
`a.remove(x)`	Elimina la primera aparición de x en la lista. Da error si x no existe
`a.pop([i])`	Eliminar el elemento en la posición i de las lista y lo devuelve. Si no se especifica, a.pop(), elimina y devuelve el último elemento

Algunos métodos de listas II

Método	Descripción
`a.index(x)`	Devuelve el índice de la primera aparición de x en la lista. Da error si x no existe
`a.count(x)`	Devuelve el número de veces que aparece x en la lista
`a.sort([cmp=, [key=]])`	Ordena los elementos de la lista en la propia lista. Se le puede pasar una función para la clave o para comparar dos elementos
`a.reverse()`	Invierte el orden de los elementos de la lista en la propia lista

Todos estos métodos son operaciones que se realizan sobre la propia lista

Ejemplos

```
>>> a = [1, -3, "cadena"]
>>> a.append(25)
>>> a.insert(0, 25)
>>> a.remove("cadena")
>>> a
[25, 1, -3, 25]

>>> a.count(25)
2
>>> a.sort()       # Si hay elementos no comparables salta un error
>>> a
[-3, 1, 25, 25]
```

Ejemplos: las variables son referencias

```
>>> a = [1, -3, "cadena"]
>>> b = a
>>> b.append(25)
>>> a
[1, -3, 'cadena', 25]
>>> c = ['otra', 'lista']
>>> a.insert(2, c)
>>> b
[1, -3, ['otra', 'lista'], 'cadena', 25]

>>> c[1] = 'listita'
>>> c
['otra', 'listita']

>>> a
[1, -3, ['otra', 'listita'], 'cadena', 25]
```

Tipo booleano

- Palabras clave para determinar verdadero y falso son **`True`** y **`False`**
- Como en C, 0 corresponde a False y cualquier otro número a True.
- De forma similar con otros objetos al convertirlos a booleano. Esto es, todo es verdadero excepto elementos *vacíos*: '', [ ], etc.

Operador	**Descripción**
`X and Y`	Devuelve True si X e Y son verdaderos
`X or Y`	Devuelve True si X o Y es verdadero
`not X`	Niega el valor de X

Algunos detalles de los operadores booleanos

- Cuando se combinan varios operadores se va evaluando de izquierda a derecha hasta que se puede determinar la salida con certeza
- X and Y devuelve Y si X es True y devuelve X si X es False
- X or Y devuelve X si X es True y devuelve Y si X es False

```
>>> True or False
True
>>> True and False
False
>>> 'hola' or ''
'hola'
>>>  'hola' and ''
''
```

Constructores de los tipos básicos

- Recuerda que en python todo son objetos.
- Cada tipo de datos tiene un constructor: int(), long(), float(), str(), bool(), complex(), list()
- Principalmente se utilizan para hacer castings y conversiones

```
>>> int("12")
12
>>> str(12)
'12'
>>> float("3.14")
3.14
>>> str ("3.14")
"3.14"
```

```
>>> bool("Hola")
True
>>> bool(0)
False
>>> complex(1, 2)
(1+2j)
>>> complex("1+3j")
(1+3j)
```

Funciones `type()` e `id()`

- Función `type()` te devuelve el tipo de un objeto:

```
>>> type(12)
int
>>> [type(True), type(3.14), type("rodaja"), type([1,2])]
[bool, float, str, list]
```

- La función `id()` te devuelve el identificador de un objeto:

```
>>> a = 2
>>> id(a)
12706112
>>> a = "cadena"
>>> id(a)
140025719526528
>>> a += "123"
>>> a
'cadena123'
>>> id(a)
140025719526672
```

Instrucciones de control: funciones

```
def nombrefuncion(argumentos):
    bloque de operaciones de la función
    return valor
```

- Elementos importantes comunes a python:
  - No hay que indicar tipos de entrada ni de salida.
  - De nuevo, el bloque de código se determina por la indentación.
  - Si no se quiere devolver ningún valor se puede dejar omitir el return o dejarlo vacío
  - Se pueden devolver múltiples valores.

```
def operations(x,y):
    return x+y, x-y, x*y, x/y
a,b,c,d=operations(3,2)
print (a,b,c,d)
k=operations(3,2)
print (k[0],k[1],k[2],k[3])
```

Instrucciones de control: funciones

```
def imprimirNombre(nombre,apellidos,reverse):
    if (not(reverse)):
        print (nombre,apellidos)
    else:
        print (apellidos,nombre)
```

- Paso de parámetros (I):
  - Llamada a la función: `imprimirNombre("Thomas","Bayes",False)`
  - El orden de los parámetros se puede variar si se usan los nombres de los mismos en la llamada a la función: `imprimirNombre(reverse=True,apellidos="Bayes",nombre="Thomas")`
  - No se pueden usar parámetros posicionales (sin indicar nombre) después de parámetros por clave (usando nombre). Ejemplo: `imprimirNombre("Thomas",apellidos="Bayes",True)`
  - Se pueden definir valores por defecto para los parámetros: `def imprimirNombre(nombre,apellidos,reverse=False)` , en cuyo caso no será necesario pasar el parámetro en la llamada a la función si se desea que tome el valor por defecto: `imprimirNombre("Thomas","Bayes")`

Instrucciones de control: funciones

- Paso de parámetros (II):
  - Se pueden definir funciones que admitan un número variable de **parámetros posicionales** (positional arguments). Estos argumentos se almacenan en una tupla (inmutable).
  - Se pueden definir funciones que admitan un número variable de **parámetros por clave** (keyword arguments), en los que cada parámetro está definido por un nombre y su valor. Estos argumentos se almacenan como un diccionario (mutable), siendo las claves del diccionario los nombres de los argumentos.
  - Más detalles sobre el paso de argumentos en Python: https://docs.python.org/3.7/tutorial/controlflow.html#more-on-defining-functions

Instrucciones de control: expresiones lambda

- Funciones **anónimas** se pueden crear mediante la palabra reservada **`lambda`**.
- Las funciones lambda están restringidas a una única expresión.
- Por ejemplo: `lambda a,b : a+b` devuelve la suma de sus dos argumentos.
- Suelen ser muy útiles cuando se quieren hacer operaciones sencillas sin necesidad de definir una función:

```
(lambda x: x > 2)(3)    # => True
(lambda x, y: x ** 2 + y ** 2)(2, 1) # => 5
```

- O para operaciones sobre listas:
  - **`map()`**: ejecuta la función definida sobre cada elemento de la lista
  - **`filter()`**: para filtrar valores de una lista de acuerdo a cierto criterio

```
map(lambda x,y: x+y, [1,2,3,4,5],[5,4,3,2,1])    # => [6, 6, 6, 6, 6]
filter(lambda x: x > 5, [3, 4, 5, 6, 7])    # => [6, 7]
```

Instrucciones de control: if

```
if condición 1:
    operaciones si se cumple la condición 1
elif condición 2:
    operaciones si se cumple la condición 2 y no la 1
else:
    operaciones si no se cumplen ni la condición 1 ni la 2
```

- Elementos importantes comunes a python:
  - Los : indican que viene un bloque de código asociado a la instrucción de control
  - El bloque de código se determina por la indentación. No hay llaves como en C o Java. Más elegante y te obliga a indentar el código

Instrucciones de control: if (ejemplo)

```
>>> edad = input("Introduzca su edad: ")
Introduzca su edad: 44
>>> if edad < 0:
...     print('Error: edad negativa')
... elif edad < 18:
...     print('Menor de edad')
... else:
...     print('Mayor de edad')
...
Mayor de edad
```

Bucle while

```
while condición:
    operaciones mientras se cumple la condición
```

- Ejemplo. Números de Fibonacci:

```
>>> a, b = 0, 1
>>> while a < 144:
....    print b, # La coma final evita imprimir el salto de línea
....    a, b = b, a+b
....
1 1 2 3 5 8 13 21 34 55 89 144
```

Bucle for

```
for elemento in secuencia:
    operaciones para elemento
```

- El bucle for sirve para recorrer secuencias.
- La secuencia debe ser algo *iterable* como cadenas, listas, tuplas, etc.
- En cada iteración del bucle *elemento* contiene un elemento de la secuencia

```
>>> s = 'hola'
>>> for c in s:
....    print c,
....
h o l a
```

Función range

```
range(stop)
range(start, stop[, step])
```

- Crea secuencias de números enteros desde `start` hasta `stop` usando el paso `step`.
- No se incluye `stop` en la secuencia.
- El paso puede ser negativo
- Se suele combinar con el bucle `for`

Función range

```
>>> range(5)
[0,1,2,3,4]


>>> range(3,8)
[3,4,5,6,7]


>>> range(1,12,2)
[1,3,5,7,9,11]


>>> range(0,-5,-1)
[0,-1,-2,-3,-4]

>>> for i in range(5):
....    print i,
0 1 2 3 4
```

Tuplas

- Son listas inmutables
- Se definen con elementos separados por comas con paréntesis () opcionales

```
>>> mitupla1 = (True, 3.14, "rodaja")  # Los paréntesis son opcionales
>>> mitupla2 = True, 3.14, "rodaja"
>>> mitupla1 + mitupla2
(True, 3.14, "rodaja", True, 3.14, "rodaja")
```

- Tuplas vacías hay que poner paréntesis

```
>>> mituplavacia = ()
>>> mituplavacia
()
```

- Tuplas de uno acaban con coma

```
>>> a = (3.14, ) # Si ponemos (3.14) sería simplemente una operación
>>> a            #  numérica y el resultado es un float y no una tupla
(3.14, )
```

Desempaquetado de secuencias

- Las tuplas, y secuencias en general, se pueden desempaquetar

```
>>> mitupla = True, 3.14, "rodaja"
>>> a, b, c = mitupla
>>> b+b
6.28
```

- Con cadenas y listas también funciona

```
>>> t = 'hola'
>>> a,b,c,d = t
>>> b + a + '!'
'oh!'
```

- Muy utilizado en la entrada y salida en funciones

Diccionarios I

- Tipo básico asociativo para almacenar pares clave-valor.
- Se representan: {claveA:valorA,...,claveZ:valorZ}
- Las claves solo pueden ser de algún tipo inmutable: cadenas, números, etc.
- La función `len` te da su longitud

```
>>> tel = {'mengano':231, 'fulano':321}
>>> len(tel)
2
>>> tel['zutano'] = 123
>>> len(tel)
3
```

Diccionarios II

- Los corchetes sirven para acceder a un valor del diccionario mediante la clave
- La palabra clave **`in`** sirve para saber si una clave está en el diccionario.

```
>>> tel = {'zutano':123, 'mengano':231, 'fulano':321}
>>> if 'zutano' in tel
....    print (tel['zutano'])
....
123
>>> print tel['utan']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: key 'utan' not found
```

Diccionarios: algunos métodos

Función	Descripción
`d.pop(key)`	Elimina la clave key del diccionario
`d.items()`	Devuelve un iterador para recorrer (clave, valor) del diccionario
`d.keys()`	Devuelve un iterador para recorrer las claves del diccionario
`d.values()`	Devuelve un iterador para recorrer los valores del diccionario

Recorriendo un diccionario

- Recorriendo claves y valores

```
>>> tel = {'zutano':123, 'mengano':231, 'fulano':321}
>>> for k,v in tel.items():
....    print (k,":",v)
....
zutano: 123 mengano: 231 fulano: 321
```

- Recorriendolo ordenado por clave

```
>>> keys_sorted = sorted(tel.keys())
>>> for k in keys_sorted:
....    print (k,":",tel[k])
....
fulano: 321 mengano: 231 zutano: 123
```

Conjuntos I

- Tipo básico para almacenar conjuntos, es decir, sin valores repetidos.
- Se representan con llaves { } con elementos separados por comas.
- El set vacío se crea con set() ya que {} es un diccionario vacío
- La función `len` te da su longitud

```
>>> fruta = {'manzanas', 'peras'}
>>> len(fruta)
2
```

Conjuntos II

- La palabra clave `in` sirve para saber si un elemento está en el conjunto.
- Los operadores funcionan como cabría esperar
  - c1 & c2: Crea un nuevo conjunto que es la intersección de c1 y c2
  - c1 | c2: Crea un nuevo conjunto que es la unión de c1 y c2
  - c1 ^ c2: Crea un nuevo conjunto que es un xor de c1 y c2
  - c1 – c2: Crea un nuevo conjunto donde con los elementos en c1 que no están en c2
  - c1 <= c2: Comprueba si c1 es un subconjunto de c2

Conjuntos: algunos métodos

Función	Descripción
`c.add(e)`	Añade un elemento al conjunto
`c.remove(e)`	Elimina el elemento e del conjunto. Salta un error si e no está
`c.discard(e)`	Como remove pero no salta un error si e no está
`c.clear()`	Elimina todos los elementos del conjunto
`c.isdisjoint(c2)`	Comprueba si todos los conjuntos c y c2 son disjuntos

Más información en:

https://docs.python.org/3.7/library/stdtypes.html#set

Ejemplos con conjuntos

- Se puede crear desde una secuencia

```
>>> a = set('zutano')
>>> b = set('mengano')
>>> b   # Solo letras no repetidas
set(['m', 'e', 'n', 'g', 'a', 'o'])
>>> a & b
set(['a', 'o', 'n'])
>>> a | b
set(['m', 'e', 'n', 'g', 'a', 'o', 'z', 'u', 't'])
>>> a - b
set(['z', 'u', 't'])
```

Ficheros

- La función **`open`** devuelve un objeto `file` (manejador de fichero) y típicamente se utiliza con dos parámetros: el nombre del fichero y el modo en que será accedido. Por ejemplo: `f=open('mifichero','w')`.
  - **`'r'`**: Solo lectura (valor por defecto si no se especifica el modo de acceso)
  - **`'w'`**: Solo escritura (si ya existe un fichero con el mismo nombre, su contenido se borrará).
  - **`'a'`**: *Appending*. Escritura al final del fichero.
  - **`'r+'`**: Para lectura y escritura.
  - **`'rb','wb','ab'`**: Para trabajar con ficheros binarios.
- Para el cierre de fichero y liberación de los recursos asociados se utiliza la función **`close`**.

Ficheros

Función	Descripción
`f.read(size)`	Lectura de los siguientes size bytes
`f.read()`	Devuelve un string con el contenido de todo el fichero
`f.readline()`	Devuelve un string con la siguiente línea en el fichero (incluye \n)
`f.readlines()`	Devuelve un lista de strings con cada línea del fichero
`f.write(string)`	Escribe la cadena pasada como parámetro en el fichero
`f.write(S)`	Escribe cada una de las cadenas en la lista S como líneas del fichero
`f.seek(offset,from_what)`	Para situar el manejador de fichero en la posición offset (las posiciones se miden en bytes) desde la posición de referencia from_what (0: inicio del fichero –por defecto-; 1: posición actual; 2: fin del fichero)
`f.tell()`	Devuelve un entero con la posición actual del manejador medida en bytes.

- Cuando los ficheros tienen un formato conocido, existen módulos que facilitan su manipulación. Por ejemplo, módulo **csv** o módulo **json**.

Ficheros

- En Python, un fichero es una secuencia de líneas, por lo que se puede iterar sobre su contenido de la siguiente forma:

```
f=fopen('filename','r')
for line in f:
    print (line)
f.close()
```

- Es una buena práctica utilizar la palabra clave **`with`** cuando se trabaja con ficheros. **`with`** hace que el fichero se cierre una vez que se ha ejecutado el bloque de código que engloba y también en caso de que se lance alguna excepción durante la ejecución.

```
with open('filename','r') as f:
    read_data=f.read()
f.closed # => True
```

Clases

Definición de la clase

```
class Human:
    species = "H. sapiens"
    def __init__(self, name):
        self.name = name
        self.age = 0
    def say(self, msg):
        return "{0}: {1}".format(self.name, msg)
    @classmethod
    def get_species(cls):
        return cls.species
    @staticmethod
    def grunt():
        return "*grunt*"
    @property
    def age(self):
        return self._age
    @age.setter
    def age(self, age):
        self._age = age
    @age.deleter
    def age(self):
        del self._age
```

Variable de clase

constructor de la clase. Solo puede haber uno.

variables de instancia

Método de la instancia

Todos los métodos toman self como primer argumento

Método de clase. Compartido por todas las instancias. Se invocan con el nombre de la clase como primer argumento.

Método estático. Se invoca sin ninguna clase o instancia.

getter

setter

Borra la variable de la instancia

Clases

```
# Instanciar la clase
i = Human(name="Ian")
print (i.say("hi"))

j = Human("Joel")
print (j.say("hello"))

# Llamada al método de la clase
i.get_species()   # => "H. sapiens"

# Se cambia el atributo de la clase
Human.species = "H. neanderthalensis"
i.get_species()   # => "H. neanderthalensis"
j.get_species()   # => "H. neanderthalensis"

# Llamanda al método estático
Human.grunt()   # => "*grunt*"

# Asignación de un nuevo valor al atributo de instancia
i.age = 42

# Se obtiene el valor del atributo
i.age # => 42

# Borrado del atributo
del i.age
i.age  # => Lanza excepción AttributeError
```

Clases: Herencia

```
class DerivedClassName(BaseClassName):
    <statement-1>
    . . .
    <statement-N>
```

- La clase hija puede sobreescribir métodos de la(s) clase(s) padre. Simplemente se redefine la función en la clase hija.
- Para acceder a la clase padre, se puede usar el propio nombre de la clase (`BaseClassName`) o la función **`super()`**
- Dos métodos built-in para trabajar con herencia:
  - **`isinstance()`** para comprobar el tipo de objeto de una instancia. Por ejemplo: `isinstance(obj,int)` devolverá `True` si `obj.__class__` es `int` o alguna clase que hereda de `int`
  - **`issubclass()`** para comprobar herencia. Por ejemplo, `issubclass(bool,int)` devuelve `True` ya que bool es una subclase de int.
- Python soporta herencia múltiple.
- Más información sobre clases en Python: https://docs.python.org/3.7/tutorial/classes.html#

Clases: Herencia

```
class Pet(object):
    def __init__(self, name, species):
        self.name = name
        self.species = species
    def getName(self):
        return self.name
    def getSpecies(self):
        return self.species
    def __str__(self):
        return "%s is a %s" % (self.name,
self.species)
```

```
class Dog(Pet):
    def __init__(self, name, chases_cats):
        Pet.__init__(self, name, "Dog")
        self.chases_cats = chases_cats
    def chasesCats(self):
        return self.chases_cats
    def getName(self):
        return super(Dog,self).getName() +
" Super Dog"
```

```
mister_pet = Pet("Mister", "Dog")
mister_dog = Dog("Mister", True)
print mister_dog.getName()    # devuelve: Mister SuperDog
print mister_dog.species      #devuelve: Dog
```

Clases Abstractas

- Python no permite definir interfaces como en Java.
- Desde Python 2.6, está disponible el **módulo abc** (Abstract Base Classes) para la definición de clases abstractas.
- Una ventaja importante es que al definir una clase que herede de la clase abstracta, se genera una excepción en caso de que exista algún método abstracto (`@abstractmethod`) sin implementar.
- En Python 3.X: **`class Base (metaclass=ABCMeta)`**

```
from abc import ABCMeta, abstractmethod
class Base(metaclass=ABCMeta):
    @abstractmethod
    def foo(self):
        pass
    def greetings(self)
        print ("hello")
```

```
class Concrete(Base):
    def foo(self):
        print ("foo")
```

Módulos

- Un módulo es un fichero fuente que contiene funciones, variables globales, clases, etc.
- El nombre del fichero que contiene el módulo viene dado por el nombre del módulo y la extensión .py
- Dentro del módulo, el nombre del módulo está accesible en la variable global __name__
- **Ejemplo**: crear el siguiente fichero llamado fibo.py en el directorio de trabajo:

```
def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print (b)
        a, b = b, a+b
def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

- Ahora desde el intérprete de Python, teclear: `import fibo`
- Usando el nombre del módulo se puede acceder a las funciones definidas en el mismo:

```
>> fibo.fib(1000)
>> fibo.fib2(100)
>> fibo.__name__
```

- La función `dir()` es usa para determinar los nombres definidos en un módulo:

```
>> dir(fibo)
```

Módulos

- Si el fichero fibo.py cambiara después de haber sido importado, deberá ser recargado mediante la instrucción `reload(fibo)`
  - Esto se puede hacer de manera automática mediante la extensión **`autoreload`**, que recarga los módulos de manera automática antes de la ejecución de código.

```
>> %load_ext autoreload
>> %autoreload 2  # 2: recarga todos los módulos (excepto los excluídos con %aimport) cada vez que se ejecuta el código
>> import fibo
>> fibo.fib(100)
1 1 2 3 5 8 13 21 34 55 89
# se cambia el código de fibo.py para que la función fib añada 1 a cada uno de los elementos de la serie de Fibonacci
>> fibo.fib(100)
2 2 3 4 6 9 14 22 35 56 90
```

- Los módulos pueden importar otros módulos. Es recomendable, pero no obligatorio, realizar los imports al comiendo del módulo.
- Ejemplo de módulo con programa main: `parrot.py`.

Importando Módulos

- Importar un módulo:

```
>> import math
>> print (math.sqrt(16))
```

- Importar funciones específicas de un módulo:

```
>> from math import sqrt
>> sqrt(16)
```

- Importar todas las funciones de un módulo. OJO! No se recomienda puesto que dificulta la comprensión de qué nombres están representados en el espacio de nombres.

```
>> from math import *
```

- Importar un módulo renombrándolo:

```
>> import math as m
>> m.sqrt(16)
```

- ¿Más información sobre módulos y paquetes: https://docs.python.org/2/tutorial/modules.html

Paquetes

- Un paquete de Python es simplemente una colección de módulos.
- Los módulos pueden ser accedidos con notación ".". Por ejemplo, A.B se refiere al módulo/subpaquete B dentro del paquete A.
- Cuando se usa `import item.subitem.subitem`, cada uno de los ítems, excepto el último, debe ser un paquete. El último ítem puede ser un módulo o un paquete, pero no una clase, variable o función.

```
>> import sound.effects.echo
>> sound.effects.echo.echofilter(input,output,delay=0.7,atten=4)
```

- También se pueden cargar módulos/paquetes como `from package import item`, donde ítem puede ser un submódulo (o subpaquete), o un nombre definido en el paquete como una función, una clase o una variable.

```
>> from sound.effects import echo
>> echo.echofilter(input,output,delay=0.7,atten=4)
>> from sounds.effects.echo import echofilter
>> echofilter(input,output,delay=0.7,atten=4)
```

Módulos y Paquetes

- Hay módulos/paquetes de Python para casi todo!
  http://wiki.python.org/moin/UsefulModules
- Algunos de los más comunes:
  - `sys,os` para tareas relaciondas con el Sistema operative
  - `math` para operaciones matemáticas estándar
  - `matplotlib` para representación gráfica
  - **`numpy`** para algebra lineal
  - `scipy` para computación científica
  - `pandas` para computación con tablas indexadas
  - **`skelearn`** para aprendizaje automático
  - `statsmodels` para estadística
  - …

NumPy

- http://docs.scipy.org/doc/numpy/contents.html
- NumPy (Numerical Python) es un paquete para computación científica básica y análisis de datos.
  - En las prácticas nos será muy útil para trabajar con vectores y matrices de forma sencilla.
- La estructura básica de NumPy son los arrays multidimensionales.
- Los arrays pueden contener elementos homogéneos de cualquier tipo.
- En NumPy, las dimensiones se denominan ejes(*axes*), y el número de *axes* se denomina rango(*rank*).
  - Por ejemplo, las coordenadas de un punto en tres dimensiones (`[1,2,1]`) es un array de rango 1 con longitud 3 en el único eje.
- Un array de NumPy es una instanciación de la clase `ndarray`. Algunos de los atributos más importantes de esta clase son:
  - **ndarray.ndim**: número de ejes (dimensiones) del array.
  - **ndarray.shape**: dimensiones del array. Es una tupla de enteros indicado la longitud de cada eje.
  - **ndarray.dtype**: objeto que describe el tipo de elementos almacenados en el array.

NumPy

```
In [1]: import numpy as np

In [2]: a = np.array([[1,2,3], [4,5,6]])

In [3]: print (a)
[[1 2 3]
 [4 5 6]]

In [4]: print (a.shape)
(2, 3)

In [5]: print (a.ndim)
2

In [6]: print (a.dtype.name)
int32

In [7]: print (type(a))
<class 'numpy.ndarray'>

In [8]: b = np.array([6,7,8])

In [9]: print (b)
[6 7 8]

In [10]: print (type(b))
<class 'numpy.ndarray'>

In [11]: print (b.shape)
(3,)

In [12]: print (b.ndim)
1
```

NumPy: Creando Arrays

- Se pueden crear arrays de diferentes formas:
  1. A partir de una lista o una tupla: `a = np.array([[1,2,3],[4,5,6]])`
  2. Array de dimensión conocida y todos los elementos a cero: `a=np.zeros((3,4))`
  3. Array de dimensión conocida y todos los elementos a uno: `a=np.ones((3,4))`
  4. Array de dimensión conocida y elementos sin inicializar a ningún valor (depende del estado de la memoria): `a=np.empty((3,4))`
  5. Array de secuencias de números: `a=np.arange(0,2,0.3)`

- Por defecto, NumPy infiere el tipo de los datos a partir de los valores de inicialización o, en el caso de las funciones `zeros`, `ones` y `empty`, el tipo por defecto es float.
  - Se puede especificar el tipo de dato como segundo parámetro en cualquiera de las funciones anteriores. Por ejemplo, `a = np.array([[1,2,3],[4,5,6]],dtype=float)`

NumPy: Operaciones Básicas

- Se aplican elemento a elemento del array.
- Se crea un nuevo array con el resultado de la operación.

```
In [1]: import numpy as np

In [2]: A = np.array( [[1,1], [0,1]], dtype=int )

In [3]: B = np.array( [[2.0,0], [3,4]], dtype=float )

In [4]: print ("A + B:")
A + B:

In [5]: print (A+B)
[[3. 1.]
 [3. 5.]]

In [6]: print ("A - B:")
A - B:

In [7]: print (A-B)
[[-1.  1.]
 [-3. -3.]]

In [8]: print ("B**2")
B**2

In [9]: print (B**2)
[[ 4.  0.]
 [ 9. 16.]]

In [10]: print ("A*B:")
A*B:

In [11]: print (A*B)
[[2. 0.]
 [0. 4.]]

In [12]: print ("Multplicación de matrices:")
Multplicación de matrices:

In [13]: print (A.dot(B))
[[5. 4.]
 [3. 4.]]
```

NumPy: Funciones Universales

- NumPy ofrece implementaciones de las funciones matemáticas más comunes como funciones estadísticas, función exponencial, funciones trigonométricas, `isnan`, `isinf`, etc.
- Estas funciones se denominan "universal functions" (`ufunc`) y son instancias de la clase `numpy.ufunc`.
- Pueden ser llamadas como `np.mean(a)` o `a.mean()`.
- Al igual que con las operaciones básicas, estas funciones operan elemento a elemento del array NumPy, y se crea un nuevo array con el resultado de la operación.
- Por defecto, estas operaciones actúan sobre todos los elementos del array, pero se puede indicar mediante el parámetro `axis` que solo se aplique la operación sobre alguna dimensión.
- Más información en: http://docs.scipy.org/doc/numpy/reference/ufuncs.html, http://docs.scipy.org/doc/numpy/reference/routines.math.html, http://docs.scipy.org/doc/numpy/reference/routines.statistics.html

NumPy: Funciones Universales

```
In [1]: import numpy as np

In [2]: a = np.random.random((2,3))

In [3]: print (a)
[[0.06932158 0.8320521  0.0582781 ]
 [0.02643114 0.62306897 0.96091379]]

In [4]: #s suma de todos los elementos

In [5]: print ("a.sum()")
a.sum()

In [6]: print (a.sum())
2.570065693224079

In [7]: # mínimo sobre todos los elementos

In [8]: print ("a.min():")
a.min():

In [9]: print (a.min())
0.026431142241509442

In [10]: # media por columnas

In [11]: print ("a.means(axis=0)")
a.means(axis=0)

In [12]: print (a.mean(axis=0))
[0.04787636 0.72756054 0.50959595]

In [13]: # desviación típica por filas

In [14]: print ("a.std(axis=1)")
a.std(axis=1)

In [15]: print (a.std(axis=1))
[0.36218565 0.38634666]

In [16]: # suma acumulada por filas

In [17]: print ("a.cumsum(axis=1)")
a.cumsum(axis=1)

In [18]: print (a.cumsum(axis=1))
[[0.06932158 0.90137368 0.95965179]
 [0.02643114 0.64950011 1.61041391]]
```

NumPy: Indexado

- Arrays de una dimensión se indexan de forma muy similar a como se realizaba en indexado en listas.
- En arrays multidimensionales, se tiene un índice por eje (`axis`). Los índices se proporcionan en una tupla separada por comas.
  - Cuando se proporcionan menos índices que el número de ejes, se considera que los índices son ":". Por ejemplo, en un array bidimensional a, `a[2,:]` es equivalente a `a[2,]`.

```
In [1]: import numpy as np

In [2]: a = np.random.random((4,3))

In [3]: print (a)
[[0.69135947 0.93639408 0.7397811 ]
 [0.36252568 0.38199432 0.31407813]
 [0.49243914 0.44120045 0.99102892]
 [0.90494318 0.01315192 0.59845506]]

In [4]: # elementos en la tercera fila, segunda columna

In [5]: print ("a[2,1]:")
a[2,1]:

In [6]: print (a[2,1])
0.44120044983591866

In [7]: # elementos en cada una de las filas de la segunda columna

In [8]: print ("a[0:5, 1]:")
a[0:5, 1]:

In [9]: print (a[0:5, 1])
[0.93639408 0.38199432 0.44120045 0.01315192]

In [10]: # equivalente a lo anterior

In [11]: print ("a[ : , 1]:")
a[ : , 1]:

In [12]: print (a[ : , 1])
[0.93639408 0.38199432 0.44120045 0.01315192]

In [13]: # todas las columnas de la segunda y tercera fila

In [14]: print ("a[1:3, : ]:")
a[1:3, : ]:

In [15]: print (a[1:3, : ])
[[0.36252568 0.38199432 0.31407813]
 [0.49243914 0.44120045 0.99102892]]
```

NumPy: Condiciones Lógicas (I)

- Las condiciones lógicas se obtienen en arrays booleanos:

```
In [1]: import numpy as np

In [2]: x = np.random.normal (0., 1., 8).reshape(4,2)

In [3]: print ("x:")
x:

In [4]: print (x)
[[ 0.29353434 -0.72998895]
 [ 0.3392862   0.47382798]
 [-0.65035918 -0.04459238]
 [-1.25016493  0.78382376]]

In [5]: ind_pos = x >= 0.

In [6]: print (type(ind_pos))
<class 'numpy.ndarray'>

In [7]: print (ind_pos.dtype.name)
bool

In [8]: print ("ind_pos:")
ind_pos:

In [9]: print (ind_pos)
[[ True False]
 [ True  True]
 [False False]
 [False  True]]

In [10]: ind_neg = x < 0.

In [11]: num_pos = ind_pos.sum()

In [12]: num_neg = ind_neg.sum()

In [13]: print ("num_pos:")
num_pos:

In [14]: print (num_pos)
4

In [15]: print ("num_neg:")
num_neg:

In [16]: print (num_neg)
4
```

NumPy: Condiciones Lógicas (II)

- Operaciones and y or:

```
In [17]: print (np.logical_and(ind_pos, ind_neg))
[[False False]
 [False False]
 [False False]
 [False False]]

In [18]: print (np.logical_or(ind_pos, ind_neg))
[[ True  True]
 [ True  True]
 [ True  True]
 [ True  True]]
```

- Para recuperar los índices que satisfacen la condición:

```
In [19]: ind_values_pos = np.nonzero(ind_pos)

In [20]: print (ind_values_pos)
(array([0, 1, 1, 3], dtype=int64), array([0, 0, 1, 1], dtype=int64))
```

NumPy

- Modificar las dimensiones de los arrays (redimensionar, concatenar): http://docs.scipy.org/doc/numpy/user/quickstart.html#shape-manipulation
- Álgebra lineal: submódulo `numpy.linalg` (diagonal de una matriz, traspuesta, inversa, diagonalización,…)
- Muchas más funcionalidades!
  - http://docs.scipy.org/doc/numpy/contents.html

Scikit-Learn

- http://scikit-learn.org/stable/
- `import sklearn`
- Librería estándar de Python para aprendizaje automático.
  - Herramientas simples y eficientes para minería y análisis de datos
  - Implementada sobre NumPy, SciPy, y matplotlib
  - Código abierto, también para uso comercial
- Contiene algunos de los algoritmos más usados en:
  - Aprendizaje supervisado: clasificación y regresión
  - Selección de modelos: búsqueda por rejilla, validación cruzada
  - Aprendizaje no supervisado: clustering
  - Preprocesado, selección de variables, reducción de dimensionalidad
- Más durante las prácticas…