In [7]:
```
import numpy as np

# np.random.rand(10,2) -> Matriz de aleatorios en dimensiones (d1, d2, dn)
# np.random.randn(10,2) -> Idem pero valores cumplen la distribución normal
# np.random.randint(low, high, size)-> Vector de enteros
# np.random.uniform(low, high, size)-> Vector de floats



print ("Valores random en dimensiones (d1, d2, dn)")
matriz = np.random.rand(2, 3, 4)
print ( matriz)
print ("Lo mismo pero valores de distribución normal")
print(np.random.randn(2, 3, 4))

print ("Enteros random : low (incl), high (excl), size")
low = 2
high = 4
size=10
print(np.random.randint(low, high, size))
```

```
Valores random en dimensiones (d1, d2, dn)
[[[0.04898954 0.8373936  0.35781203 0.96818014]
  [0.62586937 0.649848   0.16869484 0.53972205]
  [0.8778754  0.76109862 0.45441498 0.23672132]]

 [[0.94224506 0.33078926 0.73568894 0.92884834]
  [0.60755049 0.45179556 0.77161402 0.475897  ]
  [0.34454222 0.02673026 0.72291996 0.03649072]]]
Lo mismo pero valores de distribución normal
[[[ 0.68482768 -1.75542275 -1.39398658 -0.19867842]
  [ 1.46274398  0.43037534  1.13457458 -0.47046213]
  [-1.68827224  0.3510105  -1.82055307  0.78427183]]

 [[-0.17233657 -0.17081058 -0.22147554  1.56327739]
  [ 0.10417741 -1.43996845 -0.6092398  -0.37524786]
  [-1.13258954  1.36083107  0.06236846 -1.89459648]]]
Enteros random : low (incl), high (excl), size
[2 2 2 3 3 3 2 3 3 3]
```

In [8]:
```
size=10
print ("Retorna un número de floats entre [0.0, 1.0).")
print (np.random.random([size]))
print ("Retorna un número de floats entre low, to high, size")
print (np.random.uniform(2,80,10))
print ("Genera ejemplos randoms tomados de un array : a[, size, replace, p]")
print (np.random.choice([1,2,3,4,5,6],4))
```

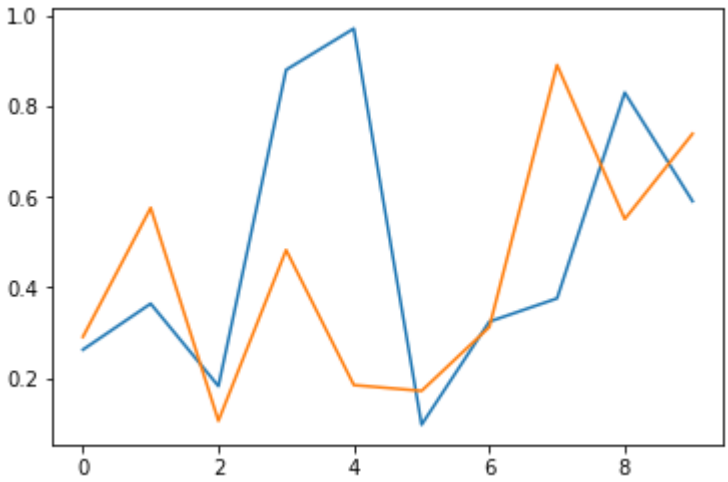
```
Retorna un número de floats entre [0.0, 1.0).
[0.26440309 0.98319285 0.23528262 0.77499521 0.87640172 0.22515587
 0.31311048 0.4845564  0.6586533  0.72781503]
Retorna un número de floats entre low, to high, size
[67.1249228  10.73877755  3.07652206 78.82191778  7.73547298 63.48218002
  2.96042873 32.53500903 24.09792351 29.92115574]
Genera ejemplos randoms tomados de un array : a[, size, replace, p]
[2 4 4 3]
```

In [9]:
```
import numpy as np

#---------------------- diccionario con una pequeña lista de temperaturas
data = {'temp': [12,13,14]}
print (data)

#---------------------- Ampliamos la lista con valores aleatorios
#data ['temp'].extend(np.random.randint(5, 12, 6))
#print (data)
```

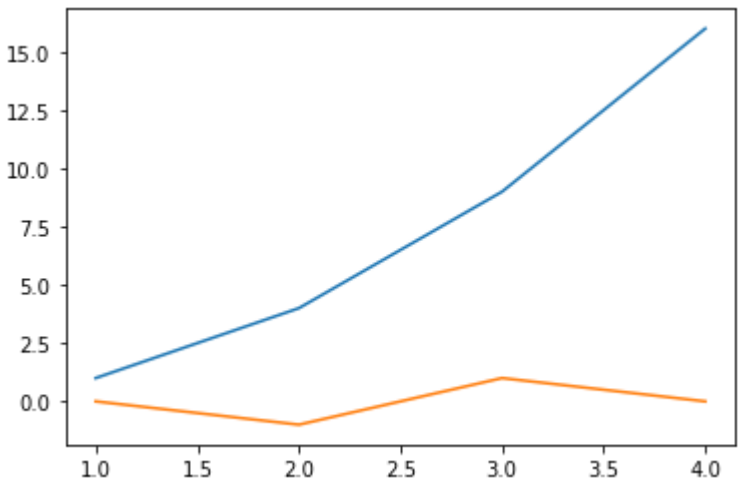
```
{'temp': [12, 13, 14]}
```

# LIBRERIA MATHPLOTLIB amb NUMPY

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.plot(np.random.rand(10))
plt.plot(np.random.rand(10))
plt.show()
```




In [6]:
```
x = [1,2,3,4]
plt.plot(x, [1,4,9,16], x,  [0,-1,1,0])
plt.show()
help(plt.plot)
```



```
Help on function plot in module matplotlib.pyplot:

plot(*args, scalex=True, scaley=True, data=None, **kwargs)
    Plot y versus x as lines and/or markers.

    Call signatures::

        plot([x], y, [fmt], *, data=None, **kwargs)
        plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)

    The coordinates of the points or line nodes are given by *x*, *y*.

    The optional parameter *fmt* is a convenient way for defining basic
    formatting like color, marker and linestyle. It's a shortcut string
    notation described in the *Notes* section below.

    >>> plot(x, y)        # plot x and y using default line style and color
    >>> plot(x, y, 'bo')  # plot x and y using blue circle markers
    >>> plot(y)           # plot y using x as index array 0..N-1
    >>> plot(y, 'r+')     # ditto, but with red plusses

    You can use `.Line2D` properties as keyword arguments for more
    control on the appearance. Line properties and *fmt* can be mixed.
    The following two calls yield identical results:

    >>> plot(x, y, 'go--', linewidth=2, markersize=12)
    >>> plot(x, y, color='green', marker='o', linestyle='dashed',
    ...      linewidth=2, markersize=12)

    When conflicting with *fmt*, keyword arguments take precedence.


    **Plotting labelled data**

    There's a convenient way for plotting objects with labelled data (i.e.
    data that can be accessed by index ``obj['y']``). Instead of giving
    the data in *x* and *y*, you can provide the object in the *data*
    parameter and just give the labels for *x* and *y*::

    >>> plot('xlabel', 'ylabel', data=obj)

    All indexable objects are supported. This could e.g. be a `dict`, a
    `pandas.DataFrame` or a structured numpy array.


    **Plotting multiple sets of data**

    There are various ways to plot multiple sets of data.

    - The most straight forward way is just to call `plot` multiple times.
      Example:

      >>> plot(x1, y1, 'bo')
      >>> plot(x2, y2, 'go')

    - Alternatively, if your data is already a 2d array, you can pass it
      directly to *x*, *y*. A separate data set will be drawn for every
      column.
```

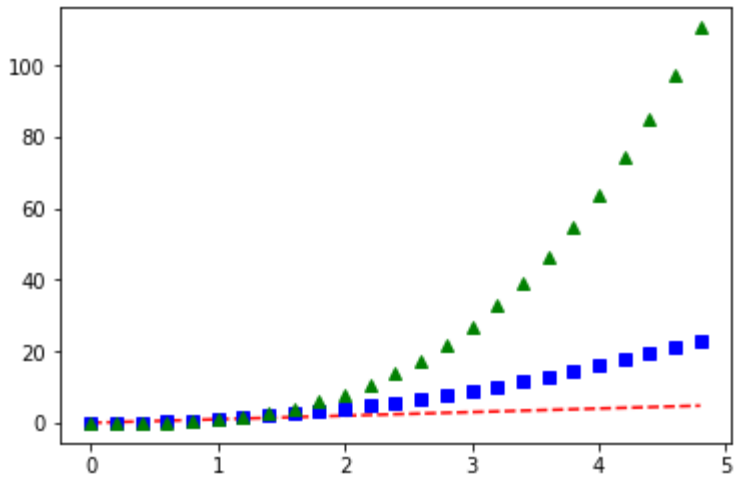
```
  Example: an array ``a`` where the first column represents the *x*
  values and the other columns are the *y* columns::

  >>> plot(a[0], a[1:])

- The third way is to specify multiple sets of *[x]*, *y*, *[fmt]*
  groups::

  >>> plot(x1, y1, 'g^', x2, y2, 'g-')

  In this case, any additional keyword argument applies to all
  datasets. Also this syntax cannot be combined with the *data*
  parameter.

By default, each line is assigned a different style specified by a
'style cycle'. The *fmt* and line property parameters are only
necessary if you want explicit deviations from these defaults.
Alternatively, you can also change the style cycle using
:rc:`axes.prop_cycle`.


Parameters
----------
x, y : array-like or scalar
    The horizontal / vertical coordinates of the data points.
    *x* values are optional and default to ``range(len(y))``.

    Commonly, these parameters are 1D arrays.

    They can also be scalars, or two-dimensional (in that case, the
    columns represent separate data sets).

    These arguments cannot be passed as keywords.

fmt : str, optional
    A format string, e.g. 'ro' for red circles. See the *Notes*
    section for a full description of the format strings.

    Format strings are just an abbreviation for quickly setting
    basic line properties. All of these and more can also be
    controlled by keyword arguments.

    This argument cannot be passed as keyword.

data : indexable object, optional
    An object with labelled data. If given, provide the label names to
    plot in *x* and *y*.

    .. note::
        Technically there's a slight ambiguity in calls where the
        second label is a valid *fmt*. ``plot('n', 'o', data=obj)``
        could be ``plt(x, y)`` or ``plt(y, fmt)``. In such cases,
        the former interpretation is chosen, but a warning is issued.
        You may suppress the warning by adding an empty format string
        ``plot('n', 'o', '', data=obj)``.

Returns
-------
list of `.Line2D`
```

```
        A list of lines representing the plotted data.

    Other Parameters
    ----------------
    scalex, scaley : bool, default: True
        These parameters determine if the view limits are adapted to the
        data limits. The values are passed on to `autoscale_view`.

    **kwargs : `.Line2D` properties, optional
        *kwargs* are used to specify properties like a line label (for
        auto legends), linewidth, antialiasing, marker face color.
        Example::

        >>> plot([1, 2, 3], [1, 2, 3], 'go-', label='line 1', linewidth=2)
        >>> plot([1, 2, 3], [1, 4, 9], 'rs', label='line 2')

        If you make multiple lines with one plot call, the kwargs
        apply to all those lines.

        Here is a list of available `.Line2D` properties:

        Properties:
        agg_filter: a filter function, which takes a (m, n, 3) float array and a dp
i value, and returns a (m, n, 3) array
        alpha: float or None
        animated: bool
        antialiased or aa: bool
        clip_box: `.Bbox`
        clip_on: bool
        clip_path: Patch or (Path, Transform) or None
        color or c: color
        contains: unknown
        dash_capstyle: {'butt', 'round', 'projecting'}
        dash_joinstyle: {'miter', 'round', 'bevel'}
        dashes: sequence of floats (on/off ink in points) or (None, None)
        data: (2, N) array or two 1D arrays
        drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post
'}, default: 'default'
        figure: `.Figure`
        fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
        gid: str
        in_layout: bool
        label: object
        linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
        linewidth or lw: float
        marker: marker style string, `~.path.Path` or `~.markers.MarkerStyle`
        markeredgecolor or mec: color
        markeredgewidth or mew: float
        markerfacecolor or mfc: color
        markerfacecoloralt or mfcalt: color
        markersize or ms: float
        markevery: None or int or (int, int) or slice or List[int] or float or (flo
at, float) or List[bool]
        path_effects: `.AbstractPathEffect`
        picker: unknown
        pickradius: float
        rasterized: bool or None
        sketch_params: (scale: float, length: float, randomness: float)
        snap: bool or None
```

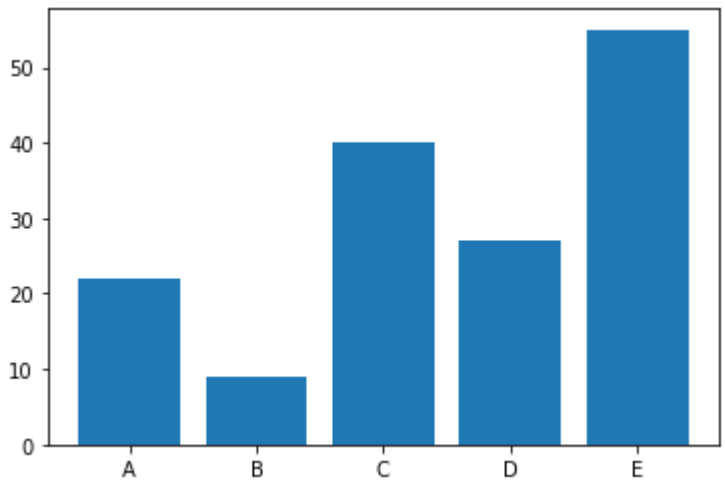
```
    solid_capstyle: {'butt', 'round', 'projecting'}
    solid_joinstyle: {'miter', 'round', 'bevel'}
    transform: `matplotlib.transforms.Transform`
    url: str
    visible: bool
    xdata: 1D array
    ydata: 1D array
    zorder: float

See Also
--------
scatter : XY scatter plot with markers of varying size and/or color (
    sometimes also called bubble chart).

Notes
-----
**Format Strings**

A format string consists of a part for color, marker and line::

    fmt = '[marker][line][color]'

Each of them is optional. If not provided, the value from the style
cycle is used. Exception: If ``line`` is given, but no ``marker``,
the data will be a line without markers.

Other combinations such as ``[color][marker][line]`` are also
supported, but note that their parsing may be ambiguous.

**Markers**

=============    ===============================
character        description
=============    ===============================
``'.'``          point marker
``','``          pixel marker
``'o'``          circle marker
``'v'``          triangle_down marker
``'^'``          triangle_up marker
``'<'``          triangle_left marker
``'>'``          triangle_right marker
``'1'``          tri_down marker
``'2'``          tri_up marker
``'3'``          tri_left marker
``'4'``          tri_right marker
``'s'``          square marker
``'p'``          pentagon marker
``'*'``          star marker
``'h'``          hexagon1 marker
``'H'``          hexagon2 marker
``'+'``          plus marker
``'x'``          x marker
``'D'``          diamond marker
``'d'``          thin_diamond marker
``'|'``          vline marker
``'_'``          hline marker
  _
=============    ===============================

**Line Styles**
```

```
=============    ================================
character        description
=============    ================================
``'-'``          solid line style
``'--'``         dashed line style
``'-.'``         dash-dot line style
``':'``          dotted line style
=============    ================================

Example format strings::

    'b'    # blue markers with default shape
    'or'   # red circles
    '-g'   # green solid line
    '--'   # dashed line with default color
    '^k:'  # black triangle_up markers connected by a dotted line

**Colors**

The supported color abbreviations are the single letter codes

=============    ================================
character        color
=============    ================================
``'b'``          blue
``'g'``          green
``'r'``          red
``'c'``          cyan
``'m'``          magenta
``'y'``          yellow
``'k'``          black
``'w'``          white
=============    ================================

and the ``'CN'`` colors that index into the default property cycle.

If the color is the only part of the format string, you can
additionally use any  `matplotlib.colors` spec, e.g. full names
(``'green'``) or hex strings (``'#008000'``).
```

In [4]:
```
import matplotlib.pyplot as plt

t = np.arange(0.0, 5.0, 0.2)
plt.plot(t,t,'r--', t,t**2, 'bs', t,t**3,'g^')
plt.show()
```




In [10]:
```
# Ejemplo de gráfico de barras

import matplotlib.pyplot as plt

x = ['A', 'B', 'C', 'D', 'E']
y = [22, 9, 40, 27, 55]

plt.bar(x, y)

plt.show()
```

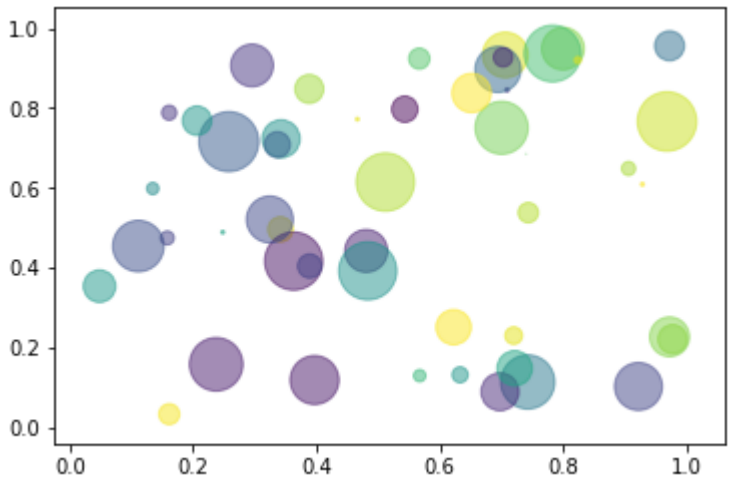



In [ ]:
```
# Práctica P01 - Ejercicio 1:
# Realiza un gráfico equivalente al del pdf con Matplotlib

A =[50, 50, 70, 67, 43, 39]
B =[40, 45, 40, 50, 50, 55]
C =[35, 25, 53, 32, 60, 50]
```

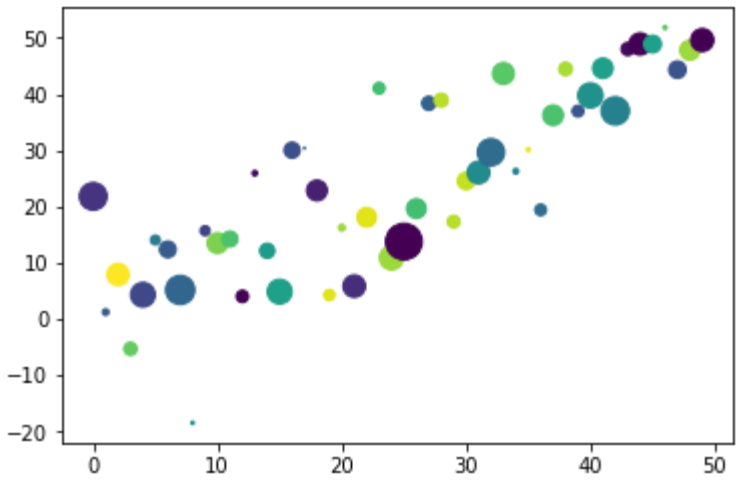
In [ ]:
```
# Práctica P01 - Ejercicio 2:
# Realiza un gráfico equivalente con Matplotlib
# Hogares equipados con PC de sobremesa en España entre 2013 y 2021
```

In [ ]:
```
# Práctica P01 - Ejercicio 3:
# Cambia los valores del programa para generar un gráfico de BARRAS equivalente al
# Gráfica de educación
x = ['A', 'B', 'C', 'D', 'E']
y = [22, 9, 40, 27, 55]
```

In [5]:
```
# EJEMPLO DE GRAFICO DE BURBUJAS

import numpy as np
import matplotlib.pyplot as plt

# Fixing random state for reproducibility
np.random.seed(19680801)


N = 50
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = (30 * np.random.rand(N))**2  # 0 to 15 point radii

plt.scatter(x, y, s=area, c=colors, alpha=0.5)
plt.show()
```




In [11]:
```
# Práctica P02 -Ejercicio 1. Estudia este código y realiza los siguientes cambios.
# Pasa de 50 elementos random a 12
# Imprime las matrices : a, b, c, d, antes de mostrar el gráfico.
# Muestra el título de cada matriz y sus valores y el gráfico del resultado

data = {'a': np.arange(50),
'c': np.random.randint(0,50,50),
'd': np.random.randn(50)}

data['b'] = data ['a'] + 10 * np.random.randn(50)
data['d'] = np.abs(data['d']) * 100

plt.scatter('a','b',c='c', s='d', data=data)
plt.xlabel = 'entry a'
plt.ylabel = 'entry b'
plt.show()
```




# Práctica P03

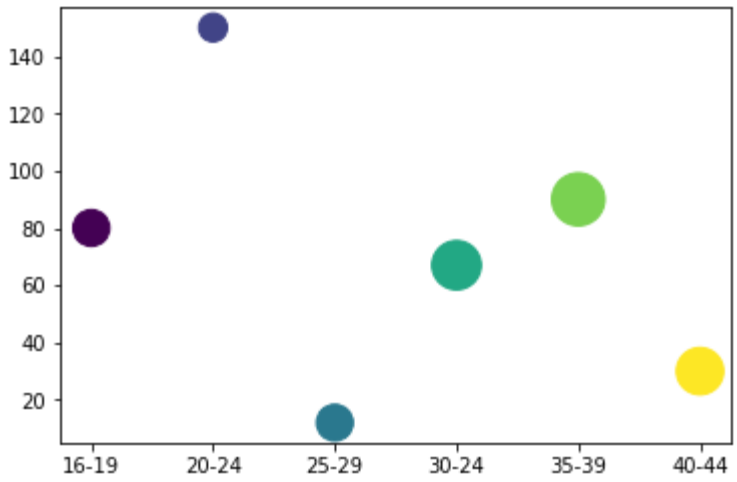
Genera con matplotlib un gráfico de dispersión (scatter) con los datos : de temperaturas medias de 12 meses del año, y volumen de lluvia mensual.

Te dan 6 meses y tienes que generar los otros 6 de forma aleatoria en un rango de valores posibles.

meses = [ 'Gen', 'Feb', 'Mar', 'Abr', 'Mayo', 'Junio'] temperaturas = [12.5, 12.3, 15.2, 16.4, 20.5, 22.2] plujamm = [335, 203, 334, 604, 700, 555]

- Tienes que mostrar 12 valores para x = (meses) y = (temperatura media)
- Medidas de los puntos, proporcionales a el volumen de lluvia cada mes
- Colores diferentes para cada mes

In [11]:
```
# OTRO EJEMPLO DE GRAFICO DE BURBUJAS
import numpy as np
import matplotlib.pyplot as plt

data = {'a': np.arange(1)}
data ['a'] = ['16-19', '20-24', '25-29', '30-24', '35-39','40-44']
data ['unidades'] = [80, 150, 12, 67, 90, 30]
data ['valor'] = [335 , 203, 334 , 604, 700, 555]
data ['c'] = [1,2,3,4,5,6]

plt.scatter('a', 'unidades', c='c', s='valor', data = data)
plt.show()
```




# P05 - Estudia este ejemplo

Diagrama de caja-bigote (box plot o box-whisker diagram).

Este es un diagrama donde se puede ver un resumen de una serie de forma rápida y sencilla. En él se representa el primer cuartil y el tercer cuartil, que son los extremos de la caja, el valor de la mediana (o segundo cuartil), que se representa mediante una línea dentro de la caja, y los extremos de la serie que no se consideran anómalos, los llamados 'bigotes', que son los valores extremos que están dentro del rango de 1.5 veces el rango intercuartílico (IQR por sus siglas en inglés, Inter Quartil Range).

Los valores que quedan fuera de este rango (suele ser 1.5xIQR), se consideran valores anómalos u 'outliers' y se representan como puntos fuera de los bigotes.

Por tanto, imaginemos que estamos representando

- la altura de los futbolistas federados,
- la altura de los jugadores de basket federados
- la altura de los gimnastas federados

Con un diagrama de caja-bigote podemos ver rápidamente como se distribuyen cada uno de estos conjuntos de datos y podemos compararlos visualmente entre ellos

In [11]:
```
import numpy as np

# Creamos unos valores para la altura de 100 futbolistas
alt_futb = np.random.randn(100)+165 + np.random.randn(100) * 10
alt_futb = np.round(alt_futb, 1)
# Creamos unos valores para la altura de 100 basquet
alt_bask = np.random.randn(100)+172 + np.random.randn(100) * 12
alt_bask = np.round(alt_bask, 1)
# Creamos unos valores para la altura de 100 gimnastas
alt_gym = np.random.randn(100)+159 + np.random.randn(100) * 9
alt_gym = np.round(alt_gym, 1)

print("futbol: ", alt_futb[0:10])
print("basket: ", alt_bask[0:10])
print("gym.  : ", alt_gym[0:10])
```

```
futbol:  [178.3 166.3 176.1 185.3 185.7 166.5 164.5 161.  171.  179.3]
basket:  [190.5 182.6 138.2 175.3 194.  155.6 165.9 177.5 150.5 175.3]
gym.  :  [153.5 157.1 157.1 151.3 154.2 151.8 164.6 161.4 139.1 164.8]
```

In [12]:
```
# Al final cogemos uno de los ejemplos generados para pasar a analizar
import matplotlib.pyplot as plt

alt_futb = [167.3, 175.8, 164.7, 155.5, 151.2, 172.9, 171.2, 149.5, 178.7, 173.8]
alt_bask = [154.4, 175.5, 178.8, 168.7, 179.4, 170.5, 185.6, 175.0, 167.1, 192.9]
alt_gym =  [170.1, 159.8, 170.7, 152.1, 159.2, 167.2, 173.7, 154.8, 148.8, 140.3]

plt.ion() # Nos ponemos en modo interactivo

# El valor por defecto para los bigotes es 1.5*IQR pero lo escribimos
# explícitamente
plt.boxplot([alt_futb, alt_bask, alt_gym], sym = 'ko', whis = 1.5)
# Colocamos las etiquetas para cada distribución
plt.xticks([1,2,3], ['Futb', 'Bask', 'Gym'], size = 'small', color ='k')
plt.ylabel(u'Altura (cm)')
```

Out[12]:
```
Text(0, 0.5, 'Altura (cm)')
```