Bioinformatics

Python Programming
Introduction to the Python Language

Sérgio Crisóstomo

2026

Departamento de Ciência de Computadores

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Contents

Dictionaries

Dictionaries

- Data structure for association tables
- Each key is associated with one (and only one) value
- Analogy: a bilingual dictionary (e.g., Portuguese–English)
- We can use as keys:
  - numbers
  - strings
  - tuples
  - combinations of the above

  (only *immutable types*)
- The order of key-value pairs may not be what you expect, since it is determined by algorithms optimized for fast element access

Example: an inventory

An inventory that associates *available quantities* to *fruits*.

Item	Quantity
bananas	25
pears	12
oranges	10

Example: an inventory (cont.)

We could represent it as a list of pairs:

```
[('bananas',25), ('oranges',10), ('pears',12)]
```

Problems:

- It is necessary to traverse the list to find the value associated with a key
- It allows duplicate keys; for example:

  ```
  [('bananas',1), ('bananas',25),
   ('oranges',10), ('pears',12)]
  ```

  Does that represent 1, 25, or 26 bananas?

Creating a dictionary

```
>>> inv = {'bananas':25,'oranges':10,'pears':12}
```

We can access values by key:

```
>>> inv['bananas']
25
>>> inv['bananas'] = inv['bananas'] + 1
>>> inv
{'bananas': 26, 'oranges': 10, 'pears': 12}
```

Creating a dictionary (cont.)

More examples:

```
>>> emails = {'Maria João':'mj@mail.pt',
              'João Pedro':'jp@mail.pt'}

>>> dirs = {'N':(0,1), 'S':(0,-1),
            'W':(-1,0), 'E':(1,0)}

>>> empty = {}        # initialize an empty dictionary
```

Some operations on dictionaries

- Get the value associated with a key (error if it doesn't exist)

  `dict[key]`

- Assign a value to a key

  `dict[key] = value`

- Test if a key exists (result: True or False)

  `key in dict`

Examples

```
>>> inv = 'bananas':25,'oranges':10,'pears':12
>>> inv['bananas']
25

>>> inv['kiwis']
KeyError: 'kiwis'

>>> 'bananas' in inv
True
>>> 'kiwis' in inv
False
```

More operations

Get the value or a *default* if the *key* doesn't exist:

```
dict.get(key, default)
```

Examples:

```
>>> inv = {'bananas':25,'oranges':10,'pears':12}
>>> inv.get('bananas',0)
25
>>> inv.get('kiwis',0)
0
```

More operations (cont.)

Iterate over all keys:

```
for key in dict.keys():
    ...
```

Example:

```
>>> inv = {'bananas':25,'oranges':10,'pears':12}
>>> for fruit in inv.keys():
...     print(fruit)

bananas
oranges
pears
```

More operations (cont.)

Get a list with all keys:

```
list(dict.keys())
```

Example:

```
>>> list(inv.keys())
['bananas', 'oranges', 'pears']
```

More operations (cont.)

Iterate over all key-value pairs:

```
for key, value in dict.items():
    ...
```

Example:

```
>>> for fruit, quant in inv.items():
...     print(fruit, quant)

bananas 25
oranges 10
pears 12
```

More operations (cont.)

Iterate over all key-value pairs sorted by key:

```
for key in sorted(dict):
    ...
```

Example:

```
>>> for fruit in sorted(inv):
...     print(fruit, inv[fruit])

bananas 25
oranges 10
pears 12
```

Counting letter occurrences

- Problem: count the occurrence of each letter of the alphabet in a text file.
- We'll use a dictionary to associate each letter with its count.
- Such a table is called a *histogram*.

Example text

Excerpt from Canto I of *Os Lusíadas*[1].

lusiadas_CantoI.txt

```
As armas e os barões assinalados,
Que da ocidental praia Lusitana,
Por mares nunca de antes navegados,
Passaram ainda além da Taprobana,
Em perigos e guerras esforçados,
...
```

[1] Source: Instituto Camões `http://www.instituto-camoes.pt`.

Histogram of occurrences

```
def histogram(file):
    f = open(file, "r")    # open file for reading
    count = {}             # initialize empty count
    while True:            # read text line by line
        line = f.readline().lower()
        if line == '':
            break
        for c in line:
            if 'a' <= c <= 'z':
                count[c] = 1 + count.get(c, 0)
    f.close()
    return count
```

Counting letter occurrences

```
>>> hist = histogram("lusiadas_CantoI.txt")
>>> hist
{'a': 3015, 'c': 658, 'b': 249, 'e': 3064,
 'd': 1217, 'g': 378, 'f': 240, 'i': 1154,
 'h': 239, 'j': 103, 'm': 1033, 'l': 572,
 'o': 2622, 'n': 1294, 'q': 390, 'p': 531,
 's': 1828, 'r': 1578, 'u': 1031, 't': 1229,
 'v': 425, 'y': 1, 'x': 29, 'z': 68}
```

Visualization

Print a table of letter occurrences:

```
hist = histogram("lusiadas_CantoI.txt")
for c in hist.keys():
    print(c, hist[c])  # letter, count
```

We can plot the histogram directly in Python with matplotlib (see next slide).

Histogram chart with matplotlib

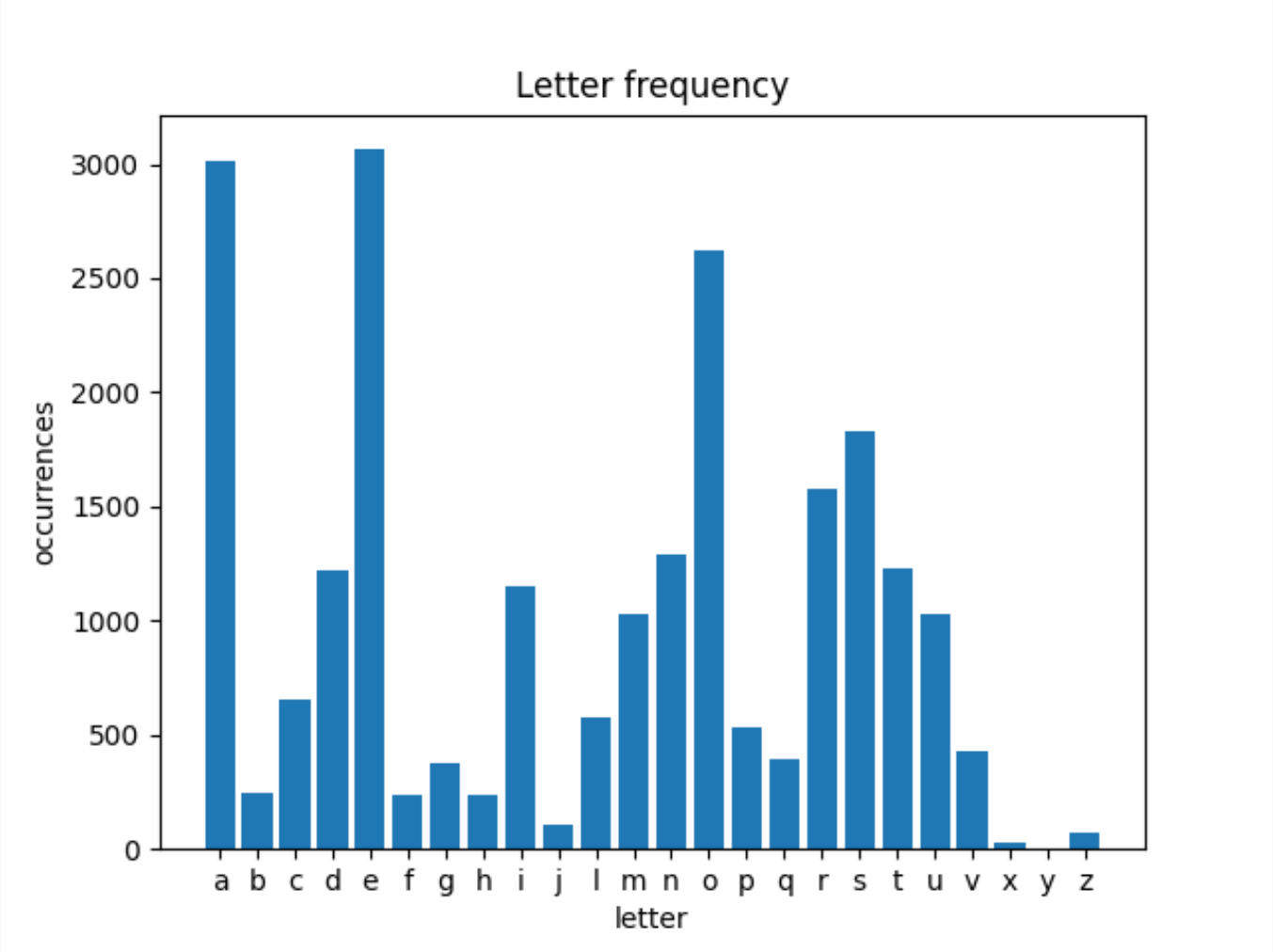
We feed the dictionary to `matplotlib` to draw a bar chart:

```
import matplotlib.pyplot as plt

def plot_histogram(hist):
    letters = sorted(hist)    # x axis: a..z
    counts = [hist[c] for c in letters]
    plt.bar(letters, counts)
    plt.xlabel("letter")
    plt.ylabel("occurrences")
    plt.title("Letter frequency")
    plt.show()
```

```
hist = histogram("lusiadas_CantoI.txt")
plot_histogram(hist)
```

Histogram chart



Observations

- The letter `'e'` occurs most frequently (3064 times)
- The letter `'y'` occurs only once
- However, the results are incorrect because accented letters are not counted
- To fix this, we will convert accented letters into non-accented ones

Normalization

In Unicode, an accented letter can be represented in two ways:

NFC a composed numeric code; e.g., ç is U+00C7 (LATIN CAPITAL LETTER C WITH CEDILLA)

NFD a pair of codes for the letter and the accent; e.g., ç is U+0043 (LATIN CAPITAL LETTER C) + U+0327 (COMBINING CEDILLA)

Normalization (cont.)

The `unicodedata` module defines a function `normalize` that allows conversion between these forms.

`normalize('NFC', str)` converts *str* to the composed form

`normalize('NFD', str)` converts *str* to the decomposed form

To ignore accents, we'll use NFD normalization:

‘ã’ ⟶ ‘a’ + ‘~’
‘á’ ⟶ ‘a’ + ‘´’
‘Ç’ ⟶ ‘C’ + ‘¸’
⋮

Histogram of occurrences (version 2)

```
from unicodedata import normalize

def histogram(file):
    f = open(file, "r")    # open file for reading
    count = {}             # initialize empty count
    while True:            # read text line by line
        line = f.readline().lower()
        if line == '':
            break
        for c in normalize('NFD', line):
            if 'a' <= c <= 'z':
                count[c] = 1 + count.get(c, 0)
    f.close()
    return count
```

Revised count

```
{'a': 3296, 'c': 739, 'b': 249, 'e': 3180,
 'd': 1217, 'g': 378, 'f': 240, 'i': 1210,
 'h': 239, 'j': 103, 'm': 1033, 'l': 572,
 'o': 2707, 'n': 1294, 'q': 390, 'p': 531,
 's': 1828, 'r': 1578, 'u': 1042, 't': 1229,
 'v': 425, 'y': 1, 'x': 29, 'z': 68}
```

The most frequent letter is `'a'`, followed by `'e'`.

Dictionaries and DNA sequences

- A DNA sequence is a string over the alphabet `{A, C, G, T}` (the four *nucleotides*)
- Dictionaries are a natural fit for two common tasks:
  - counting how often each nucleotide occurs
  - using a codon table to translate DNA into a protein
- Both reuse the same idea as the letter histogram

Counting nucleotides

The same pattern as the letter histogram, restricted to A/C/G/T:

```
def nucleotide_counts(dna):
    counts = {}
    for base in dna:
        counts[base] = 1 + counts.get(base, 0)
    return counts
```

```
>>> dna = "ATGGCCATTGGCCGCTAA"
>>> nucleotide_counts(dna)
{'A': 4, 'T': 4, 'G': 5, 'C': 5}
```

Translating codons with a codon table

A codon is a group of 3 nucleotides that codes for one amino acid.
We store the (partial) genetic code in a dictionary:

```
codon_table = {
    'ATG':'M', 'GCC':'A', 'ATT':'I',
    'GGC':'G', 'CGC':'R', 'TAA':'*',
    'TAG':'*', 'TGA':'*',
}   # '*' marks a STOP codon
```

Translating codons with a codon table (cont.)

We read the sequence three letters at a time and look each codon up in the table (using a default for unknown codons):

```
def translate(dna):
    protein = ""
    for i in range(0, len(dna) - 2, 3):
        codon = dna[i:i+3]
        protein += codon_table.get(codon, '?')
    return protein
```

```
>>> translate("ATGGCCATTGGCCGCTAA")
'MAIGR*'
```

`.get(codon, '?')` returns `'?'` for any codon that is missing from our partial table.

Summary

- Dictionaries represent association tables
- The order of entries is not significant
- Searching by key is more efficient than searching through a list of pairs

Sets

Sets

- A set is an unordered collection of distinct (unique) elements
- Like the mathematical notion of a set
- Key properties:
  - no duplicates – repeated elements are kept only once
  - no order – elements have no position or index
  - very fast membership test with `in`
- Elements must be of an *immutable type* (numbers, strings, tuples) – the same restriction as dictionary keys

Creating sets

Use braces `{...}` or the `set()` function:

```
>>> s = {3, 1, 2, 2, 1}    # duplicates removed
>>> s
{1, 2, 3}
>>> set([1, 1, 2, 3])      # from a list
{1, 2, 3}
>>> set("ACGTA")           # from a string
{'A', 'C', 'G', 'T'}
>>> empty = set()     # NOT {} -- that is a dict!
```

Note: since sets are unordered, the display order is arbitrary.

Membership, adding and removing

```
>>> bases = {'A', 'C', 'G', 'T'}
>>> 'A' in bases
True
>>> 'U' in bases
False
>>> bases.add('U')          # add an element
>>> bases.discard('T')      # remove if present
>>> len(bases)              # number of elements
4
```

Membership testing (`in`) is much faster on a set than on a list.

Set operations

Sets support the usual mathematical operations:

```
>>> a = {1, 2, 3, 4}
>>> b = {3, 4, 5, 6}
>>> a | b       # union
{1, 2, 3, 4, 5, 6}
>>> a & b       # intersection
{3, 4}
>>> a - b       # difference
{1, 2}
>>> a ^ b       # symmetric difference
{1, 2, 5, 6}
```

Why sets? Finding distinct elements

A set is the simplest way to get the distinct values in a sequence:

```
>>> dna = "ATGGCCATTGGCCGCTAA"
>>> set(dna)            # distinct nucleotides
{'A', 'C', 'G', 'T'}
>>> len(set(dna))       # how many distinct ones
4
```

- Use a dictionary when you need to associate a *value* with each key (e.g. a count)
- Use a set when you only care about *which* elements are present

Plotting with Matplotlib

Plotting with Matplotlib

- matplotlib is the most widely used plotting library in Python
- The `pyplot` submodule offers a simple interface, imported by convention as:
  - `import matplotlib.pyplot as plt`
- Typical workflow:
  1. build the plot with calls such as `plt.plot`, `plt.bar`, ...
  2. add labels and a title
  3. display it with `plt.show()`, or save it with `plt.savefig(...)`

A first plot

A line plot from two lists of coordinates:

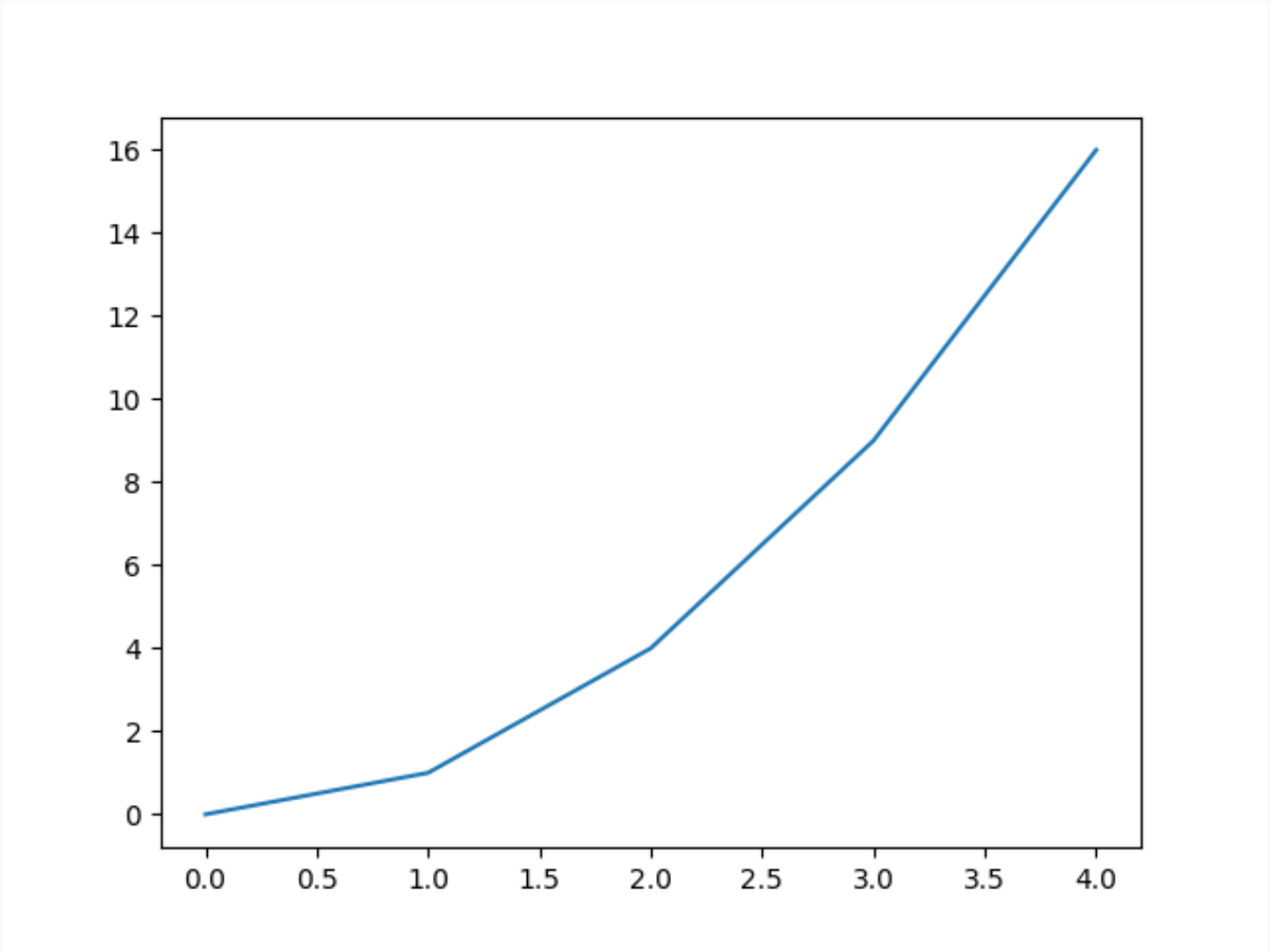
```
import matplotlib.pyplot as plt

x = [0, 1, 2, 3, 4]
y = [0, 1, 4, 9, 16]

plt.plot(x, y)
plt.show()
```

`plt.plot(x, y)` draws a line through the points `(x[0], y[0])`, `(x[1], y[1])`,...

A first plot

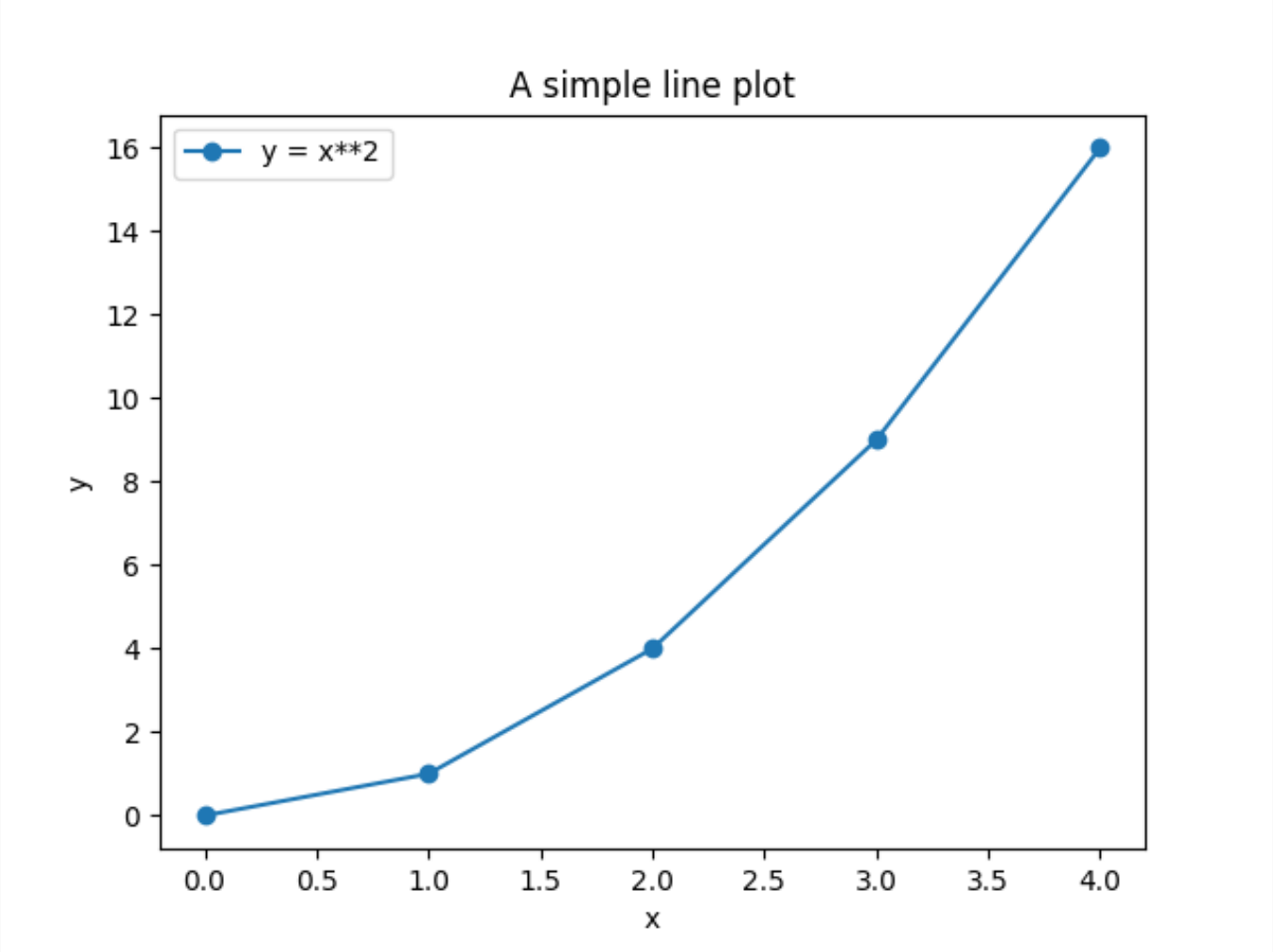


Labels, title and legend

```
plt.plot(x, y, marker="o", label="y = x**2")
plt.xlabel("x")
plt.ylabel("y")
plt.title("A simple line plot")
plt.legend()
plt.show()
```

- `marker="o"` draws a circle at each data point
- `label=...` together with `plt.legend()` shows a legend

Labels, title and legend



Bar charts

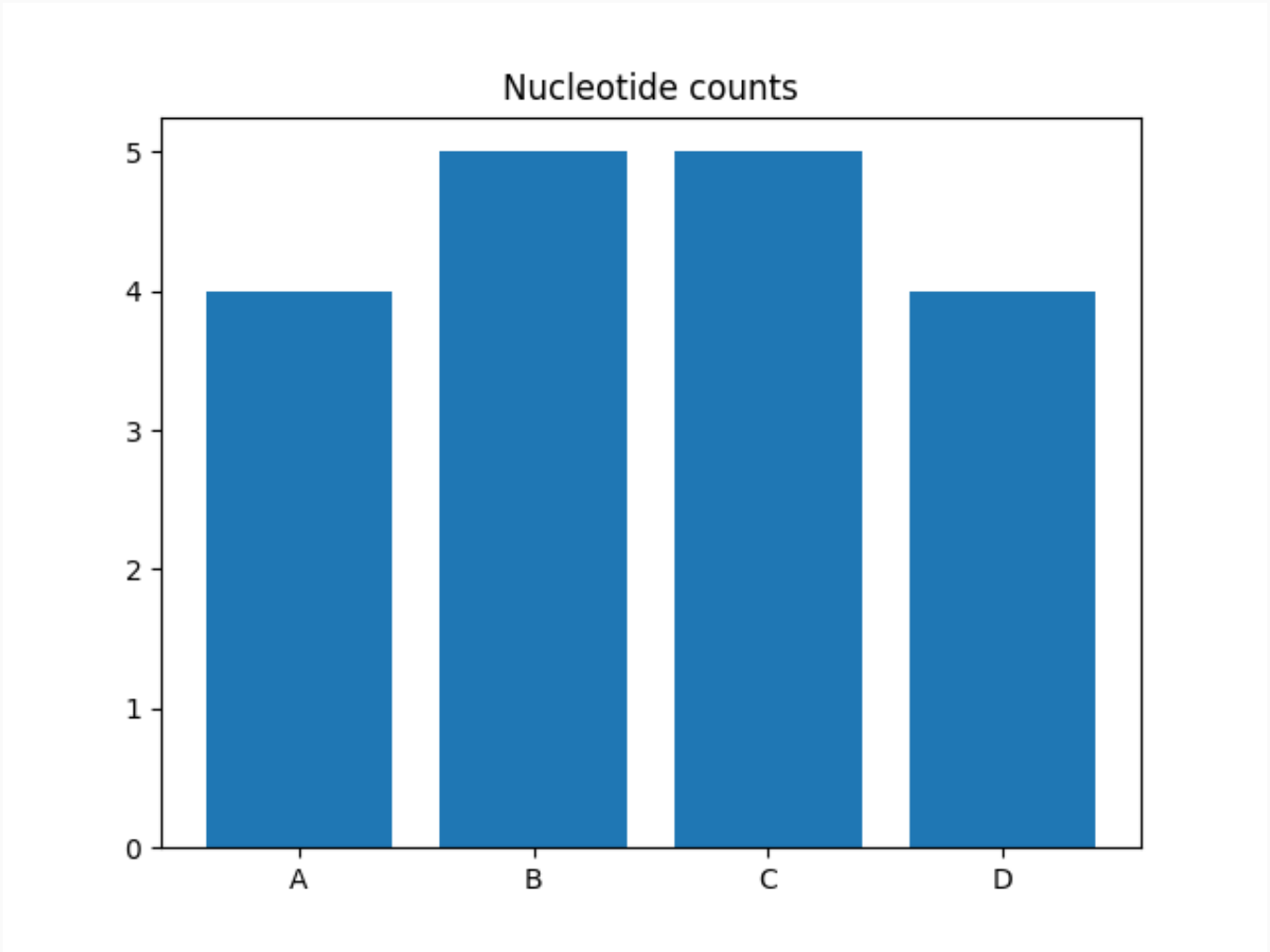
`plt.bar` takes the categories and their values:

```
labels = ["A", "C", "G", "T"]
values = [4, 5, 5, 4]

plt.bar(labels, values)
plt.title("Nucleotide counts")
plt.show()
```

This is exactly the kind of chart we used to visualise the letter (and nucleotide) histograms.

Bar charts



Other plot types and saving figures

- `plt.scatter(x, y)` – a cloud of points
- `plt.hist(data)` – a histogram from raw data
- `plt.savefig("file.png")` – save to a file instead of opening a window

```
plt.scatter([1, 2, 3], [2, 1, 3])
plt.savefig("points.png")    # save instead of show
```