Python Programming for Data Science

Week 36, Friday

- Types:
  - lists, tuples, dictionaries
  - files

Recap: datatypes — a popquiz

- How many different numeric types are there in Python?
- What are booleans used for?
- How do you create an empty string?
- What does "\n" mean?
- What does the `*` do for strings?
- Can you add a string to a number?
- How do you find the length of a string?
- How do you extract a substring from a string?
- What is the difference between a function and a method?
- What does the `strip` string-method do?

Lists

Types: Lists

Lists are a *collection* type. They allow you to gather several values and manipulate them as one.

Lists are ordered collections of items, accessible by an index.

- lines from a file
- multiple sequence alignment
- results from a series of experiments
- ...

A list is created using square brackets:

```
l = [1, 2.0, "three", 4]    # Lists can contain values of different types
l = [1,2,3,4,5,6]
```

Types: Lists — indexing

Lists and strings are both *sequence types*, and have similar properties. Indexing works exactly as for strings:

```
l = [1,2,3,4,5,6]
print(l[2])
```

output

```
3
```

Types: Lists — indexing

Lists and strings are both *sequence types*, and have similar properties. Indexing works exactly as for strings:

```
l = [1,2,3,4,5,6]
print(l[2])
```

output

```
3
```

One difference is that in lists, you can *assign* to an index:

```
l[2] = 5
print(l)
```

output

```
[1, 2, 5, 4, 5, 6]
```

Types: Lists — indexing

Lists and strings are both *sequence types*, and have similar properties. Indexing works exactly as for strings:

```
l = [1,2,3,4,5,6]
print(l[2])
```

```
output
3
```

One difference is that in lists, you can *assign* to an index:

```
l[2] = 5
print(l)
```

```
output
[1, 2, 5, 4, 5, 6]
```

You cannot do this to strings (they are *immutable*).

Types: Lists — operators

Lists have exactly the same operators as strings:

```
+, +=, *, *=, [], ==, !=, <, <=, >, >=, in
```

Examples:

```
l = [1,2,3,4]
print(l + [5,6])
l += [5,6]
print(l)
print(3 in l)
print(l > [5,6])
print([5,6] * 3)
```

output

```
[1, 2, 3, 4, 5, 6]

[1, 2, 3, 4, 5, 6]
True
False
[5, 6, 5, 6, 5, 6]
```

Types: Lists — methods

Commonly used list methods:

`append`	add element to the end of a list
`count`	Count occurrences of value
`index`	Return index of first occurrence of value
`insert`	Insert element at specified index
`pop`	Remove and return element from (end of) list
`remove`	remove first occurrence of specified value
`reverse`	reverse order of elements in list (in place)
`sort`	sort elements in list (in place)

Examples:

```
l = [1,2,3,4,5,6]
l.append(7)
l.reverse()
print(l)
```

output

```
[7, 6, 5, 4, 3, 2, 1]
```

Types: Strings & lists: slicing

Lists and strings are both *sequence* types

Sequence types support *slicing* to extract subsequences:

```
sequence[m:n]
```

This returns a sequence with the elements from index m up to n-1. (note: endpoint not included)

```
s = "hello"
print(s[1:4])
```

output

```
'ell'
```

Types: Strings & lists: slicing

Lists and strings are both *sequence* types

Sequence types support *slicing* to extract subsequences:

```
sequence[m:n]
```

This returns a sequence with the elements from index m up to n-1. (note: endpoint not included)

```
s = "hello"
print(s[1:4])
```

```
output
'ell'
```

Optionally, you can provide a third argument to a slice, specifying a step-size:

```
print(s[1:5:3])
```

```
'eo'
```

output

Types: Strings & lists: slicing (2)

Range parameters can be omitted, indicating that everything in that direction should be included:

```
s = "hello"
print(s[:4])
print(s[1:])
print(s[:])
```

output

```
'hell'
'ello'
'hello'
```

Types: Strings & lists: slicing (2)

Range parameters can be omitted, indicating that everything in that direction should be included:

```
s = "hello"
print(s[:4])
print(s[1:])
print(s[:])
```

```
output

'hell'
'ello'
'hello'
```

And for lists:

```
l = [1,2,3,4,5]
print(s[:4])
print(s[1:])
print(s[:])
```

output

```
[1,2,3,4]
[2,3,4,5]
[1,2,3,4,5]
```

Lists — exercise

1. Create a variable containing the following string:

```
Horse sense is the thing a horse has which keeps it
from betting on people. W.C. Fields
```

2. Convert this string to a list of words.
3. Find the index of the word ”Horse” in the sequence.
4. Sort the list.
5. Print out the first 3 elements of this sorted list.
6. Bonus exercise: Try to reverse the list of words using the slice operation.

Lists — exercise — solution

1. Create a variable containing the string:

```
quote = '''Horse sense is the thing a horse has which keeps it
... from betting on people. W.C. Fields'''
```

2. Convert this string to a list of words.

3. Find the index of the word ”Horse” in the sequence.

Lists — exercise — solution

1. Create a variable containing the string:

```
quote = '''Horse sense is the thing a horse has which keeps it
... from betting on people. W.C. Fields'''
```

2. Convert this string to a list of words.

```
word_list = quote.split()
print(word_list)
```

```
                                                                                out
['Horse', 'sense', 'is', 'the', 'thing', 'a', 'horse', 'has', 'which',
```

3. Find the index of the word ”Horse” in the sequence.

Lists — exercise — solution

1. Create a variable containing the string:

```
quote = '''Horse sense is the thing a horse has which keeps it
... from betting on people. W.C. Fields'''
```

2. Convert this string to a list of words.

```
word_list = quote.split()
print(word_list)
```

```
                                                                                                                                out
['Horse', 'sense', 'is', 'the', 'thing', 'a', 'horse', 'has', 'which',
```

3. Find the index of the word ”Horse” in the sequence.

```
print(word_list.index("Horse"))
```

```
0                                                                                                                               out
```

Lists — exercise — solution (2)

4. Sort the list.

```
word_list.sort()
```

5. Print out the first 3 elements of this sorted list.

6. Bonus exercise: Can you come up with a way to use slicing to reverse the order of the word list

```
print(word_list[::-1])
```

Lists — exercise — solution (2)

4. Sort the list.

```
word_list.sort()
```

5. Print out the first 3 elements of this sorted list.

```
print(word_list[:3])
```

```
output
['Fields', 'Horse', 'W.C.']
```

6. Bonus exercise: Can you come up with a way to use slicing to reverse the order of the word list

```
print(word_list[::-1])
```

Tuples

Types: Tuples

Tuples are immutable lists. This means that you cannot alter them in any way after you have created them (just like strings).

Types: Tuples

Tuples are immutable lists. This means that you cannot alter them in any way after you have created them (just like strings).

Tuples are defined using commas, and often using parentheses:

```
t = (1,2,3,4)
t = 1,2,3,4     # parenthesis can be ommitted
```

Types: Tuples

Tuples are immutable lists. This means that you cannot alter them in any way after you have created them (just like strings).

Tuples are defined using commas, and often using parentheses:

```
t = (1,2,3,4)
t = 1,2,3,4     # parenthesis can be ommitted
```

Example:

```
x = 3
y = 4
print((x,y))    # Making a tuple on-the-fly
```

output

```
(3, 4)
```

Types: Tuples — assigning

You can assign to multiple variables at once by using a tuple on the left-hand side of the assignment

```
my_tuple = (1,2)     # Creating a tuple
x, y = my_tuple      # Assigning variable x to 1 and variable y to 2
```

Types: Tuples — assigning

You can assign to multiple variables at once by using a tuple on the left-hand side of the assignment

```
my_tuple = (1,2)    # Creating a tuple
x, y = my_tuple     # Assigning variable x to 1 and variable y to 2
```

Another example:

```
x,y = y,x      # Nice trick: swapping the values
print((x,y))   # of x and y
```

```
output

(2, 1)
```

Types: Tuples — operators

Tuples support a subset of the operators and methods of lists (only ones that don't modify the collection):

```
+, *, [], ==, !=, <, <=, >, >=, in
```

Types: Tuples — operators

Tuples support a subset of the operators and methods of lists (only ones that don't modify the collection):

```
+, *, [], ==, !=, <, <=, >, >=, in
```

Just as with strings, you cannot assign to an entry in a tuple (due to immutability):

```
t = (1,2,3,4)
t[2] = 1
```

output

```
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: object doesn’t support item assignment
```

Types: Tuples — why?

Why have tuples when we already have lists?

Types: Tuples — why?

Why have tuples when we already have lists?

- Tuples are smaller and faster.

Types: Tuples — why?

Why have tuples when we already have lists?

- Tuples are smaller and faster.
- Tuples are used as return values from functions.

Types: Tuples — why?

Why have tuples when we already have lists?

- Tuples are smaller and faster.
- Tuples are used as return values from functions.
- The fact that tuples are immutable makes it possible to use them as keys in a dictionary.

Types: Tuples — why?

Why have tuples when we already have lists?

- Tuples are smaller and faster.
- Tuples are used as return values from functions.
- The fact that tuples are immutable makes it possible to use them as keys in a dictionary.

Tuples are a streamlined version of lists. If you know in advance that your sequences are fixed (e.g. coordinates in space) then use tuples.

Tuples — exercise

1. Create a tuple called `triplet` of size 3 containing the numbers 1, 2, and 3.
2. Create three variables `x`, `y` and `z` and use the tuple `triplet` to initialize them.
3. Now create a tuple called `quartet` of size 4 (with the numbers 1,2,3,4), and try it again. What happens?
4. Tuples support slicing. Can you use this to fix the problem?

Tuples — exercise — solution

1. Create a tuple called `triplet` of size 3 containing the numbers 1, 2, and 3.

```
triplet = (1,2,3)
```

2. Create three variables `x`, `y` and `z` and use the tuple `triplet` to initialize them.

```
x,y,z = triplet
```

3. Now create a tuple called `quartet` of size 4, and try it again. What happens?

Tuples — exercise — solution

1. Create a tuple called `triplet` of size 3 containing the numbers 1, 2, and 3.

```
triplet = (1,2,3)
```

2. Create three variables `x`, `y` and `z` and use the tuple `triplet` to initialize them.

```
x,y,z = triplet
```

3. Now create a tuple called `quartet` of size 4, and try it again. What happens?

```
quartet = (1,2,3,4)
x,y,z = quartet
```

output

```
Traceback (most recent call last):
  File "/home/lpp/lpp2016/test.py", line 2, in <module>
    x,y,z = quartet
ValueError: too many values to unpack
```

Tuples — exercise — solution (2)

1. Tuples support slicing. Can you use this to fix the problem?

```
x,y,z = quartet[:3]     # Only use index 0,1,2
```

Tuples — exercise — solution (2)

1. Tuples support slicing. Can you use this to fix the problem?

```
x,y,z = quartet[:3]     # Only use index 0,1,2
```

or

```
x,y,z,_ = quartet       # _ is often used as a dummy variable
```

Dictionaries

Types: Dictionaries

Recall: lists give you access to a value through their index:

```
my_list = ["a", "b", "c"]
print(my_list[1])     # lookup using index. Index 1 => "b"
```

Types: Dictionaries

Recall: lists give you access to a value through their index:

```
my_list = ["a", "b", "c"]
print(my_list[1])     # lookup using index. Index 1 => "b"
```

Dictionaries are similar, but more general: they give access to values by their *key*, which can for instance be strings.

Types: Dictionaries

Recall: lists give you access to a value through their index:

```
my_list = ["a", "b", "c"]
print(my_list[1])    # lookup using index. Index 1 => "b"
```

Dictionaries are similar, but more general: they give access to values by their *key*, which can for instance be strings.

Dictionaries consist of (key, value) pairs. They are defined using curly brackets with ":" as key/value separator:

Types: Dictionaries

Recall: lists give you access to a value through their index:

```
my_list = ["a", "b", "c"]
print(my_list[1])    # lookup using index. Index 1 => "b"
```

Dictionaries are similar, but more general: they give access to values by their *key*, which can for instance be strings.

Dictionaries consist of (key, value) pairs. They are defined using curly brackets with ":" as key/value separator:

Example:

```
days = {'Mon':'Monday', 'Tue':'Tuesday', 'Wed':'Wednesday'}
```

Idea: associate weekdays with their abbreviations

Types:Dictionaries — accessing elements

You can look up an element in a dictionary using []:

```
days = {'Mon':'Monday', 'Tue':'Tuesday', 'Wed':'Wednesday'}
print(days['Mon'])
```

output

```
'Monday'
```

Types:Dictionaries — accessing elements

You can look up an element in a dictionary using []:

```
days = {'Mon':'Monday', 'Tue':'Tuesday', 'Wed':'Wednesday'}
print(days['Mon'])
```

output

```
'Monday'
```

Note the similarity to lists, except that individual elements are not associated with an index number, but a *key*.

Types:Dictionaries — accessing elements

You can look up an element in a dictionary using []:

```
days = {'Mon':'Monday', 'Tue':'Tuesday', 'Wed':'Wednesday'}
print(days['Mon'])
```

output

```
'Monday'
```

Note the similarity to lists, except that individual elements are not associated with an index number, but a *key*.

You will get a KeyError if you use a key that doesn’t exist:

```
print(days['Fri'])
```

output

```
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    print(days['Fri'])
KeyError: 'Fri'
```

Types: Dictionaries — inserting and deleting elements

You can add new elements to a dictionary using the [] operator:

```
days = {}    # empty dictionary
days['Mon'] = 'Monday'
print(days)
```

output

```
{’Mon’: ’Monday’}
```

Types: Dictionaries — inserting and deleting elements

You can add new elements to a dictionary using the [] operator:

```
days = {}    # empty dictionary
days['Mon'] = 'Monday'
print(days)
```

output

```
{’Mon’: ’Monday’}
```

Elements can be deleted using the `del` statement:

```
del days['Mon']
print(days)
```

output

```
{}
```

Types: Dictionaries are not ordered

One important difference from lists, is that dictionaries are not ordered

```
days = {'Mon':'Monday', 'Tue':'Tuesday', 'Wed':'Wednesday'}
print(days)
```

```
{'Tue': 'Tuesday', 'Mon': 'Monday', 'Wed': 'Wednesday'}
```

Types: Dictionaries are not ordered

One important difference from lists, is that dictionaries are not ordered

```
days = {'Mon':'Monday', 'Tue':'Tuesday', 'Wed':'Wednesday'}
print(days)
```

```
{'Tue': 'Tuesday', 'Mon': 'Monday', 'Wed': 'Wednesday'}
```

Take home message: You cannot rely on the dictionary maintaining the order in which you insert elements - you can only access them by their key.

Types: Dictionaries are not ordered

One important difference from lists, is that dictionaries are not ordered

```
days = {'Mon':'Monday', 'Tue':'Tuesday', 'Wed':'Wednesday'}
print(days)
```

```
{'Tue': 'Tuesday', 'Mon': 'Monday', 'Wed': 'Wednesday'}
```

Take home message: You cannot rely on the dictionary maintaining the order in which you insert elements - you can only access them by their key.

This has changed in recent versions of Python (3.7+). They are now ordered by order of insertion.

Types: Dictionaries — operators

The only really meaningful operators for dictionaries are

```
==, !=, [], in
```

which provide the same functionality as they do in lists

Types: Dictionaries — methods

Some interesting dictionary methods:

`get`	Similar to `[]` - without key errors
`items`	Return list of (key,value) tuples
`keys`	Return list of keys
`pop`	Remove specified key and return value
`popitem`	Remove specified key and return (key,value) tuple
`setdefault`	Same as get, but add (key, None) if key is not already there
`update`	Add all elements from specified dictionary to current dictionary
`values`	Return list of values

```
>>> days.keys()
['Tue', 'Mon', 'Wed']
>>> days.pop('Wed')
'Wednesday'
>>> 'Wed' in days
False
```

Dictionaries — exercise

1. Create a dictionary with the keys "firstname", "lastname", and "age", and appropriate values.
2. Add a key named "address" to this dictionary.
3. Print out the list of keys in your dictionary.
4. Create a "name" key which as value contains a string with both your first and last names. Then remove the first name and last name keys. Can you do this without actually typing in your name again?

Dictionaries — exercise — solution

1. Create a dictionary with the keys firstname, lastname, and age, and appropriate values.

```
personal_info = {"firstname": "Donald",
                 "lastname": "Trump",
                 "age": 74}
```

2. Add an `address` key to this dictionary.

```
personal_info["address"] = "Whitehouse"
```

3. Print out the list of keys in your dictionary.

Dictionaries — exercise — solution

1. Create a dictionary with the keys firstname, lastname, and age, and appropriate values.

```
personal_info = {"firstname": "Donald",
                 "lastname": "Trump",
                 "age": 74}
```

2. Add an `address` key to this dictionary.

```
personal_info["address"] = "Whitehouse"
```

3. Print out the list of keys in your dictionary.

```
print(personal_info.keys())
```

```
['lastname', 'age', 'firstname', 'address']
```
output

Note the different order.

Dictionaries — exercise — solution (2)

4. Create a "name" key which as associated value contains a string with both your first and last names. Then remove the first name and last name keys. Can you do this without actually typing your name again?

```
personal_info["name"] = personal_info["firstname"] + " " + \
                        personal_info["lastname"]
del personal_info["firstname"]
del personal_info["lastname"]
personal_info
{'age': 74, 'name': 'Donald Trump', 'address': 'Whitehouse'}
```

Dictionaries — exercise — solution (2)

4. Create a "name" key which as associated value contains a string with both your first and last names. Then remove the first name and last name keys. Can you do this without actually typing your name again?

```
personal_info["name"] = personal_info["firstname"] + " " + \
                        personal_info["lastname"]
del personal_info["firstname"]
del personal_info["lastname"]
personal_info
{'age': 74, 'name': 'Donald Trump', 'address': 'Whitehouse'}
```

Or:

```
personal_info["name"] = personal_info.pop("firstname") + " " + \
                        personal_info.pop("lastname")
```

The NoneType

Types - the None value

It is possible to specify an empty value – the value None.

None is often returned from functions as an empty result:

```
print(days.get('bla'))  # Get returns None if the key is not found
```

output

```
None
```

Files

The file type

The file type is used to represent a file in a program

You can open a file using the open function:

```
open(filename, mode)
```

where mode can be:

```
'r'     - read
'w'     - write
'r+w'   - read and write
'a'     - append
```

The open function returns a File object. Remember to save this object to a variable.

```
input = open('.bashrc', 'r')
output = open('outputfile.txt', 'w')
```

Files - methods

Commonly used methods:

`readline`	Read one line as a string
`readlines`	Read file as a list of strings
`write`	Write string to file
`writelines`	Write list to file (as lines)
`close`	Close file

Example:

```
input = open('/home/lpp/.bashrc', 'r')
lines = input.readlines()
print(lines[0])
input.close()
```

```
"# ~/.bashrc: executed by bash(1) for non-login shells.\n"
```

output

Files - methods

Commonly used methods:

`readline`	Read one line as a string
`readlines`	Read file as a list of strings
`write`	Write string to file
`writelines`	Write list to file (as lines)
`close`	Close file

Example:

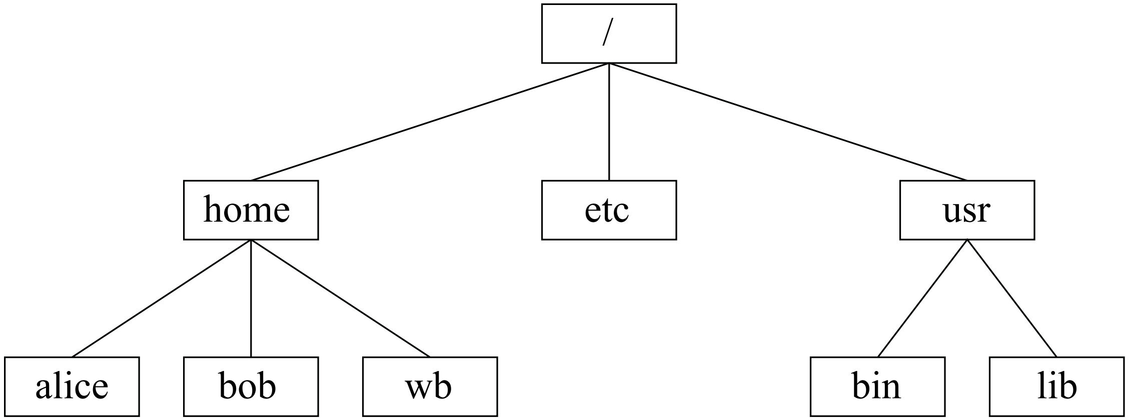
```
input = open('/home/lpp/.bashrc', 'r')
lines = input.readlines()
print(lines[0])
input.close()
```

output

```
"# ~/.bashrc: executed by bash(1) for non-login shells.\n"
```

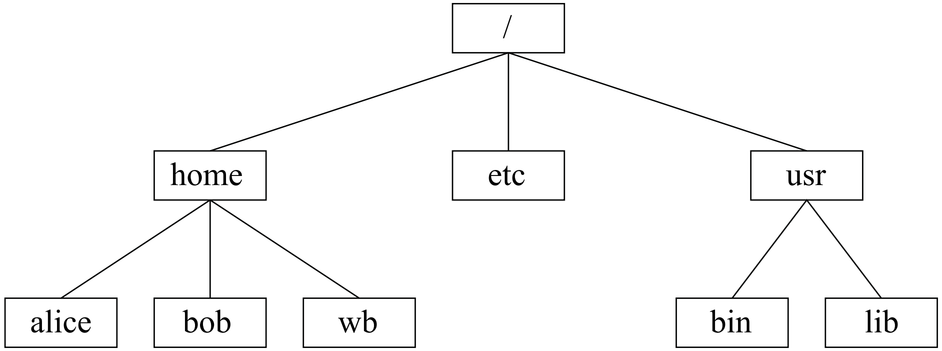
What does `/home/lpp/.bashrc` mean?

Files and Directories



- Files are organized similar to Windows, OSX, ...
- The *current directory* is where you are in the tree right now.
- The root directory is the directory at the highest level of the file system hierarchy.
  Unix/Linux: `/`
  Windows: Name of drive - e.g. `C:\`

Absolute and Relative paths



Directories (folders) and files can be referred to by using either an **absolute** or a **relative** path:

absolute:

`/home/wb/exam_results.html` (Unix)
`C:\Users\wb\exam_results.html` (Windows)

relative: (assuming I am in `/home`)

`wb/exam_results.html` (Unix)
`wb\exam_results.html` (Windows)

Types - Files are streams

Important: you can only read a file once.

The file is a stream, you *consume* the stream when you read it

```
input_file = open('.bashrc', 'r')
file_as_str = input_file.read()
file_as_str = input_file.read()           # file_as_str will now be empty
```

Types - Files are streams

Important: you can only read a file once.

The file is a stream, you *consume* the stream when you read it

```
input_file = open('.bashrc', 'r')
file_as_str = input_file.read()
file_as_str = input_file.read()           # file_as_str will now be empty
```

There are ways to reset a file stream, so you can read it again...

Types - Files are streams

Important: you can only read a file once.

The file is a stream, you *consume* the stream when you read it

```
input_file = open('.bashrc', 'r')
file_as_str = input_file.read()
file_as_str = input_file.read()           # file_as_str will now be empty
```

There are ways to reset a file stream, so you can read it again...

...but the simplest (and most efficient) is to just read once, and save the information that you need into a variable.

Files — exercise

1. Create a new program called `file_test.py`. In this program, use the `open` function to open the `file_test.py` itself.
2. Read its content into a list of strings
3. Print the last two lines to screen

Files — exercise — solution

1. Create a new program called `file_test.py`. In this program, use the open function to open the `file_test.py` itself.

```
my_file = open("file_test.py")
```

file_test.py

2. Read its content into a list of strings

```
lines = my_file.readlines()
```

file_test.py

3. Print the last two lines to screen

```
print(lines[-2:])
```

file_test.py

Collection types: which should I use?

Collection types – an overview

We covered various collection types last week, but when do you use what?

- lists
- tuples
- dictionaries

It can be very important what collection type you choose for your data.

Here are some questions you could ask yourself:

Collection types – lists vs. dictionaries

Is my data unordered or ordered?

Examples:

Collection types – lists vs. dictionaries

Is my data unordered or ordered?

Examples:

- The lines in a file are often naturally ordered, and most efficiently stored in a list.

Collection types – lists vs. dictionaries

Is my data unordered or ordered?

Examples:

- The lines in a file are often naturally ordered, and most efficiently stored in a list.
- Experimental results on different genes might be unordered, and could be stored in a dictionary using gene-names as keys.

Collection types – lists vs. dictionaries

Is my data unordered or ordered?

Examples:

- The lines in a file are often naturally ordered, and most efficiently stored in a list.
- Experimental results on different genes might be unordered, and could be stored in a dictionary using gene-names as keys.

Note: Sometimes it makes sense to use a dictionary to store ordered data - when the data is sparse.

Collection types – lists vs. tuples

Is the collection size naturally fixed?

Tuples should be used when the type of data you are working with always has the same number of items, and the items don’t need to be changed individually

Examples:

- 3D-coordinates
- Each line of a file containing experimental results with a fixed number of columns