
python
TM

Data processing

A brief recap

At this point we know how to:

1. Organize the data in memory, with lists and lists of lists, if needed
2. Loop through the data, write conditional code, define functions
3. Read data from a system file, and write them back to another file.

Python provides a range of advanced methods to process the data: typically you load data in memory because you have to do something with them…

The data can be provided in many different formats.

Hands-on example

- Suppose we have four numerical series available, coming from anything you can think of… They can be for example the athletic performance of four different persons on a 600m race, or the results of four different measurements of a sensor, performed by four scientists.
  For our Python class it doesn't matter: we have four named series.

- You can download them from:

```
http://bit.do/anna-txt
http://bit.do/giulia-txt
http://bit.do/rosa-txt
http://bit.do/sonia-txt
```

- Our goal will be to find for each series the three shortest times.

- anna.txt
  2-34,3:21,2.34,2.45,3.01,2:01,2:01,3:10,2-22
- giulia.txt
  2.59,2.11,2:11,2:23,3-10,2-23,3:10,3.21,3-21
- rosa.txt
  2:22,3.01,3:01,3.02,3:02,3.02,3:22,2.49,2:38
- sonia.txt
  2:58,2.58,2:39,2-25,2-55,2:54,2.18,2:55,2:55

It seems that the records are separated by a comma; the notation is human readable but not consistent.

I know what you are thinking…

Mariner Bugs Out (1962)
Cost: $18.
Disaster: T
diverted fr
destroyed
Cause: A p
computer
function in
velocity as
rocket off

Mars Climate Crasher (1998)
Cost: $125 million
Disaster: After a 286-day journey from Earth, the Mars Climate Orbiter fired its engines to push into orbit around Mars. The
t's
used

Cancer Treatment to Die For (2000)
Cost: Eight people dead, 20 critically injured
Disaster: Radiation therapy software by Multidata Systems International miscalculated the proper dosage, exposing patients to harmful and in some cases fatal levels of radiation. The physicians, who were legally required to double-check the software's calculations, were indicted for murder.
Cause: The software calculated radiation dosage based **on the order in which data was entered**, sometimes delivering a double dose of radiation.

Let's load each series in a separate list

```
>>> with open('anna.txt') as anf:
        data=anf.readline()
>>> anna=data.strip().split(',')
>>> print anna
```

The previous code build a list named "anna".

Remember that you can use **`type()`** to check the data type of a variable.

Note the ***method chaining***: the first method `strip()` is applied to the string contained in `data`, the second method `split()` is applied to the resulting string, and the final result is assigned to the identifier on the left.

So you must read chained methods **from the left to the right**.

Now that we have the list, we must sort it.
We have two options:

1. Sort the list "on site", losing the original list
2. Sort a copy of the list, keeping the original one

```
>>> data = [6,3,1,2,4,5]
>>> data.sort()
>>> data
[1, 2, 3, 4, 5, 6]

>>> data=[6,3,1,2,4,5]
>>> data2=sorted(data)
>>> data
[6, 3, 1, 2, 4, 5]
>>> data2
[1, 2, 3, 4, 5, 6]
>>> data3=sorted(data, reverse=True)
>>> data3
[6, 5, 4, 3, 2, 1]
```

Sort the original list

Generate a copy

This code allows us to see a important property of Python functions.

If you pass the arguments **without names**, they are identified by their **position**.

If in the function definition the arguments have **names** and default **values**, you can pass to the function only the arguments needed, with their names.

The definition of sorted is in fact:

```
sorted(iterable, cmp=None, key=None, reverse=False)
```

http://bit.do/TimeSeries1-py

```
>>> with open('anna.txt') as anf:
        data=anf.readline()
>>> anna=data.strip().split(',')
>>> print sorted(anna)
['2-22', '2-34', '2.34', '2.45', '2:01', '2:01', '3.01', '3:10', '3:21']
```

Well, they doesn't seem much sorted…The data format, human readable, it's not suitable to be sorted by a BIF. For the sort function every list item is a string, so it uses the character order (- < . < :).

You need to write a function that takes a string as the input, formatted as the list items (minutes+separator+seconds), and returns the same string with a point as a separator between minutes and seconds.

```
>>> def sanitize(time_string):
        if '-' in time_string:
            splitter= '-'
        elif ':' in time_string:
            splitter=':'
        else:
            return time_string
        (mins,secs) = time_string.split(splitter)
        return mins + '.' + secs
```

Now you must apply the function to the lists.

You cannot apply the function to the whole list; probably the simplest way to proceed is:

1. Create a new list
2. Loop over the old list
3. Apply to each item the sanitize function
4. Add the resulting item to the new list

http://bit.do/TimeSeries2-py

```
>>> with open('anna.txt') as anf:
        data=anf.readline()
>>> anna=data.strip().split(',')
>>> clean_anna=[]                                  #1
>>> for each_t in anna:                            #2
        clean_anna.append(sanitize(each_t))        #3 - 4
>>> print sorted(clean_anna)
```

Please note the ***function chaining***: you apply a series of functions to the data, each function processes them and returns the results to the next one.

You must read the function chaining **from the right to the left**.

With Python you can also build a new list starting from an old one with a peculiar construct: the list comprehension

List comprehension

- The list comprehensions are a very common way to build new lists starting from existing lists.

```
newlist = [expression for name in list]
```

- `name` is an identifier that loops over each item of `list`; for each item `expression` is computed, and the resulting item is added to the new list, that is eventually returned to the name on the left

- `expression` must be a valid calculation for **every** item in `list`.

- If `list` contains other containers, then `name` can be a containers of names that must match the `list` items

```
>>> li=[('a',1),('b',2),('c',7)]
>>> [n*3 for (x,n) in li]
[3,6,21]
```

```
newlist = [expression for name in list]
```

`expression` can also be a **user defined function**!

```
>>> def subtract(a,b):
        return a-b
>>> list= [(6,3),(1,7),(5,5)]
>>> [subtract(y,x) for (x,y) in list]
[-3,6,0]
```

The following expression defines a **filtered list comprehension**:

```
newlist = [expression for name in list if filter]
```

`Filter` determines if the `expression` is applied on each item of `list`. If the `filter` condition is not verified for a given `name` (that is it returns `false` or `0`), that particular item is omitted from the list comprehension.

Filtered list comprehension example:

```
>>> li=[3,6,2,7,1,9]
>>> [elem * 2 for elem in li if elem >4]
[12,14,18]
```

Since the list comprehensions take a list as input and produce a list as output, they can be easily ***nested***:

```
>>> li=[3,2,4,1]
>>> [elem*2 for elem in [item +1 for item in li]]
[8,6,10,4]
```

The internal list returns [4,3,5,2], the external one returns [8,6,10,4].

Use with caution…

```
>>> mins = [1,2,3]
>>> secs = [m*60 for m in mins]
[60, 120, 180]

>>> meters = [1,10,3]
>>> feet = [m*3.281 for m in meters]
[3.281, 32.81, 9.843]

>>> lower = ['non','mangio','ostriche!']
>>> upper = [s.upper() for s in lower]
['NON', 'MANGIO', 'OSTRICHE!']

>>> dirty = ['2-22', '2:22', '2.22']
>>> clean = [sanitize(t) for t in dirty]
['2.22', '2.22', '2.22']
>>> clean = [float(s) for s in clean]      #reassignment
[2.22, 2.22, 2.22]
```

Remember that the expression in the list comprehension is always applied on each list item, and not on the entire list. You cannot use `sorted()` in the list comprehension.

Now you can rewrite the last code using `sanitize()` and the list comprehensions.

http://bit.do/TimeSeries3-py

```
def sanitize(time_string):
    if '-' in time_string:
        splitter= '-'
    elif ':' in time_string:
        splitter=':'
    else:
        return time_string

    (mins,secs) = time_string.split(splitter)
    return mins + '.' + secs

with open('anna.txt') as anf:
    data=anf.readline()

anna=data.strip().split(',')

print sorted([sanitize(t) for t in anna])
```

The inspection of the final results shows us that the lists contain **duplicates**.
We do not want duplicates, because we are searching the three best times, so how can we address this issue?
We cannot use the list comprehensions, because we can rule out a double entry only AFTER the list comprehension has finished building the list.

Take the last code, and instead of print it, reassign the result of sorted funtion to the original list like:

```
anna=sorted([sanitize(t) for t in anna])
```

You must now create a new list, and append to it only the non-duplicated items of the original list. You should also print the shortest three times.

```
unique_anna=[]
for each_t in anna:
    if each_t not in unique_anna:
        unique_anna.append(each_t)

print unique_anna[0:3]
```

http://bit.do/TimeSeries4-py

This last list processing introduces a lot of duplicated code…but as often happens in Python, there is a better way.
The Python **set** is a data structure, similar to the list, that does not allow duplicates. You can build it with the BIF **set()**, or directly with the braces **{}**, or *starting from a list, using the same BIF.*
So you can replace the code above with a single line:

```
print sorted(set([sanitize(t) for t in anna]))[0:3]
```

http://bit.do/TimeSeries5-

```
def sanitize(time_string):
    if '-' in time_string:
        splitter= '-'
    elif ':' in time_string:
        splitter=':'
    else:
        return time_string
    (mins,secs) = time_string.split(splitter)
    return mins + '.' + secs

with open('anna.txt') as anf:
    data=anf.readline()
anna=data.strip().split(',')

with open('giulia.txt') as gif:
    data=gif.readline()
giulia=data.strip().split(',')

with open('sonia.txt') as sof:
    data=sof.readline()
sonia=data.strip().split(',')

with open('rosa.txt') as rof:
    data=rof.readline()
rosa=data.strip().split(',')

print sorted(set([sanitize(t) for t in anna]))[0:3]
print sorted(set([sanitize(t) for t in giulia]))[0:3]
print sorted(set([sanitize(t) for t in sonia]))[0:3]
print sorted(set([sanitize(t) for t in rosa]))[0:3]
```

Any comment? How we can improve this code?

It lacks exception handling

There is a lot of repeated code

write another version, defining a function that eliminates the repetition of same code, and handles the exceptions.

http://bit.do/TimeSeries6-py

```
def sanitize(time_string):
    if '-' in time_string:
        splitter= '-'
    elif ':' in time_string:
        splitter=':'
    else:
        return time_string
    (mins,secs) = time_string.split(splitter)
    return mins + '.' + secs

def get_run_data(filename):
    try:
        with open(filename) as f:
            data=f.readline()
        return data.strip().split(',')
    except IOError as ioerr:
        print 'File Error: ' + str(ioerr)
        return None

anna=get_run_data('anna.txt')
giulia=get_run_data('giulia.txt')
rosa=get_run_data('rosa.txt')
sonia=get_run_data('sonia.txt')

print sorted(set([sanitize(t) for t in anna]))[0:3]
print sorted(set([sanitize(t) for t in giulia]))[0:3]
print sorted(set([sanitize(t) for t in rosa]))[0:3]
print sorted(set([sanitize(t) for t in sonia]))[0:3]
```

Advanced Data Structures

Dictionaries

- The lists allow a large number of operations on the data, but they are not suitable for every situation, that's why Python provides more advanced data structures, called **dictionaries**.
- A dictionary registers a correspondence (a mapping) between a set of **keys** and a set of **values**, building a better match between the data structure in memory and the actual data structure.
- In other programming languages the dictionaries are known as *mappings*, *hashes* or *associative arrays*.
- **The dictionaries are MUTABLE and NOT ordered data collections**.

About dictionaries

- **Keys** can be of any type, as long as is an IMMUTABLE one.
- **Values** can be of any type, *including lists and dictionaries*
- A single dictionary can contain *different types* of values.
- You can define, edit, view, search, and delete key-value pairs in the dictionary.
- In the dictionary it is not maintained any information about the element location: the keys provide the symbolic location, not physical, of the items of a dictionary

Initialization

You can define many elements in a sigle instruction:

```
>>> d1 = { 'key1' : 'value1', 'key2' : 'value2' }
>>> d2 = dict(key1=value1, key2=value2
```

You can build a dictionary element by element (dynamic growth)

```
>>> d3={}
>>> d3['key1'] = 'value1'
>>> d3['key2'] = 'value2'
>>> d3['key3'] = 'value3'
```

You can iterate over the elements (always remember that the order is not stored)

```
>>> for key in d3:
        print "d3['%s']=%s" % (key, d3[key])
```

However, you can sort the list of keys:

```
>>> for key in sorted(d3):
        print "d3['%s']=%s" % (key, d3[key])
```

Copy and assignment

The dictionaries behave like lists with reguard to copy and assignment.

```
>>> a = dict(q=6, error=None)
>>> b = a
>>> a['r'] = 2.5
>>> a
{'q': 6, 'r': 2.5, 'error': None}
>>> a is b
True
>>> a = 'a string'              # make a refer to a new (string) object
>>> b                           # new contents in a do not affect b
{'q': 6, 'r': 2.5, 'error': None}
>>> a is b
False
```

What if you want a copy?

```
>>> a = dict(q=6, error=None)
>>> b = a.copy()
>>> b is a
False
>>> b['q'] = 7
>>> a['q']                      # not affected by assignment to b['q']
6
```

Example

- Let's take the previous example, assuming that the data is now changed and contain not only time, but also the name and date of birth of the person who made these records, allowing us to identify her.
  Our goal is always to extract the data and print out the three best times for each name.

- You can download them from:

  ```
  http://bit.do/anna2-txt
  http://bit.do/giulia2-txt
  http://bit.do/rosa2-txt
  http://bit.do/sonia2-txt
  ```

- As you can see in each file there is a defined data structure, that you can logical translate into a dictionary whose keys are "name", "DOB", "Times", and values the name, date of birth and the list of times.

- anna2.txt
  Anna Magnani,
  2002-3-14,2-34,3:21,2.34,2.45,3.01,2:01,2:01,3:10,2-2
  2,2-01,2.01,2:16
- giulia2.txt
  Giulia Sagramola,
  2002-8-17,2.59,2.11,2:11,2:23,3-10,2-23,3:10,3.21,3-2
  1,3.01,3.02,2:59
- rosa2.txt
  Rosa Aulente,
  2002-2-24,2:22,3.01,3:01,3.02,3:02,3.02,3:22,2.49,2:38
  ,2:40,2.22,2-31
- sonia2.txt
  Sonia Gandhi,
  2002-6-17,2:58,2.58,2:39,2-25,2-55,2:54,2.18,2:55,2:5
  5,2:22,2-21,2.22

This is our last code (TimeSeries6), that now we'll edit to cope with the new data files.

http://bit.do/TimeSeries6-py

```
def sanitize(time_string):
    if '-' in time_string:
        splitter= '-'
    elif ':' in time_string:
        splitter=':'
    else:
        return time_string
    (mins,secs) = time_string.split(splitter)
    return mins + '.' + secs

def get_run_data(filename):
    try:
        with open(filename) as f:
            data=f.readline()
        return data.strip().split(',')
except IOError as ioerr:
        print 'File Error: ' + str(ioerr)
        return None

anna=get_run_data('anna.txt')
giulia=get_run_data('giulia.txt')
rosa=get_run_data('rosa.txt')
sonia=get_run_data('sonia.txt')

print sorted(set([sanitize(t) for t in anna]))[0:3]
print sorted(set([sanitize(t) for t in giulia]))[0:3]
print sorted(set([sanitize(t) for t in rosa]))[0:3]
print sorted(set([sanitize(t) for t in sonia]))[0:3]
```

First of all, we must modify the print functions at the end, because now the list returned by `get_run_data` does not contains only time values.

http://bit.do/Dict1-py

```
def sanitize(time_string)          #unmodified
def get_run_data(filename):        #unmodified

anna=get_run_data('anna2.txt')

(anna_name, anna_dob) = anna.pop(0), anna.pop(0)

print anna_name+"'s fastest times are: " + str(sorted(set([sanitize(t) for
t in anna]))[0:3])
```

The function `get_run_data` turns the data into a list

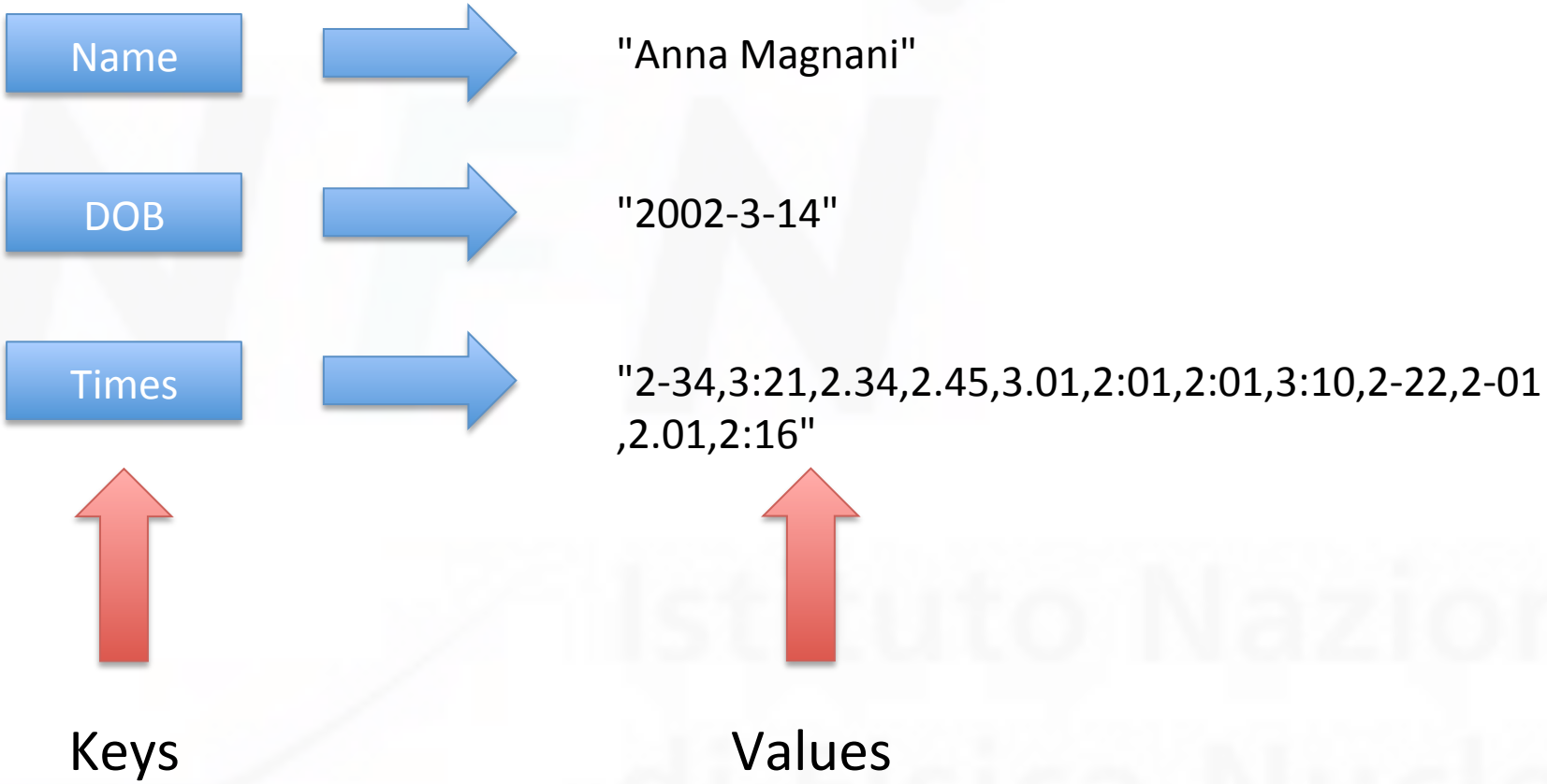
The `pop(0)` call returns and removes data from the front of a list.
Two calls to `pop(0)` remove the first two data values and assigns them to the named variables

A custom message within the call to `print` is used to display the results you're after.

This program works as expected, except that you have to name and create Anna's three variables in such as way that it's possible to identify *which* name, date of birth, and timing data relate to Anna. If you look the Dict1 file you'll see that we have 12 variables…
What if there are 40, 400, or 4,000 files to process?

anna2.txt
Anna Magnani,2002-3-14,2-34,3:21,2.34,2.45,3.01,2:01,2:01,3:10,2-22,2-01,2.01,2:16

The data have a definite logical structure: Name, Date of Birth, list of times.



This lets your in-memory data closely match the structure of your actual data.

Now we do not need anymore the code that extracts the variables from the list, and we can build the dictrionary that maps in memory the structure of our data:

```
def sanitize(time_string)          #unmodified
def get_run_data(filename):        #unmodified

anna=get_run_data('anna2.txt')

anna_data ={}
anna_data['Name'] = anna.pop(0)
anna_data['DOB'] = anna.pop(0)
anna_data['Times'] = anna
print anna_data['Name'] + "'s fastest times are: " +
str(sorted(set([sanitize(t) for t in anna_data['Times']]))[0:3])
```

http://bit.do/Dict2-py

The difference is that you can now more easily determine and control which identification data **associates** with which timing data, because they are *stored in a single dictionary*.
Any idea on how we can improve the code?

Rather than building the dictionary as you go along, why not do it all in one go? In fact, in this situation, it might even make sense to do this processing within the `get_run_data()` function and have the function return a populated dictionary.
You might also want to consider moving the `sorted()` code into the `get_run_data()` function, too, because doing so would rather nicely abstract away these processing details.

Now let's adjust the Dict2.py code following this suggestions:

- Create the dictionary all in one go
- Move the dictionary creation code into the `get_run_data()` function, returning a dictionary as opposed to a list.
- Move the code that determines the top three times for each athlete into the `get_run_data()` function.
- Adjust the invocations within the main code to the new version of the `get_run_data()` function to support it's new mode of operation.

http://bit.do/Dict3-py

```
def sanitize(time_string)          #unmodified

def get_run_data(filename):
    try:
        with open(filename) as f:
            data=f.readline()
        templ=data.strip().split(',')
        return {'Name':templ.pop(0),
                'DOB': templ.pop(0),
                'Times': str(sorted(set([sanitize(t) for t in templ]))[0:3])}
    except IOError as ioerr:
        print 'File Error: ' + str(ioerr)
        return None

anna=get_run_data('anna2.txt')

print anna['Name'] + "'s Fastest Times are: " + anna['Times']
```

1. `templ` is a temporary list that holds the data BEFORE the creation of the dictionary
2. `get_run_data(filename)` now returns a dictionary built between brackets **{ }**
3. The **code** that determines the top three scores is part of the function, too
4. To process a data file all you need is two lines of code: the first invokes the `get_run_data()` function and the second invokes `print`.

Now we are using a dictionary to keep our data all in one place, but what if we need to process them (for example compute the mean)? We must write a custom function, that is NOT associated with the data but works **only** on them. That's why we introduce classes.

Classes

Classes

Like the majority of other modern programming languages, Python lets you create and define an object-oriented **class** that can be used to *associate code with the data that it operates on*.

Using a class helps reduce complexity.

- By associating your code with the data it works on, you reduce complexity as your code base grows.

Reduced complexity means fewer bugs.

- Reducing complexity results in fewer bugs in your code. However, it's a fact of life that your programs will have functionality added over time, which will result in additional complexity. Using classes to **manage** this complexity is a *very good thing*.

Fewer bugs means more maintainable code.

- Using classes lets you keep your code and your data together in one place, and as your code base grows, this really can make quite a difference.

- Once a class definition is in place, you can use it to create (or *instantiate*) **data objects**, which inherit their characteristics from your class.

- Within the object-oriented world, your code is often referred to as the class's **methods**, and your data is often referred to as its **attributes**. Instantiated data objects are often referred to as **instances**.

- Each object is **created from** the class and **shares** a similar set of characteristics. The methods (your code) are the *same* in each instance, but each object's attributes (your data) *differ* because they were created from your raw data.

Use `class` to define classes

Python uses the keyword **`class`** to define classes. Every defined class has a *special method* called `__init__()`, (note the double underscore) which allows you to control how objects are *initialized*. Methods within your class are defined in much the same way as functions, that is, using **`def`**. Here's the basic form:

```
class Athlete:
    def __init__(self):
        #Initialization code…
```

Do not forget the colon, and the indentation that defines the code block.

`__init__` it's **always** the initialization method (constructor): the arguments passed to the istance name during the object creation are passed to this method.

Creating object instances

With the class in place, it's easy to create object instances. Simply assign a *call* to the class name to each of your variables. In this way, the class (together with the `__init__()` method) provides a mechanism that lets you create a **custom factory function** that you can use to create as many object instances as you require:

```
a=Athlete()
b=Athlete()
c=Athlete()
```

The brackets tell Python to create a new "Athlete" object, which is then assigned to the variable on the left.

Unlike in C++-inspired languages, Python has no notion of defining a constructor called "new."

The importance of `self`

- When you define a class you are, in effect, defining a *custom factory function* that you can then use in your code to create

```
a=Athlete()
```

- When Python processes this line of code, it turns the factory function call into the following call, which identifies the *class*, the *method* (which is automatically set to `__init__()`), and the

```
Athlete.__init__(a)
```

```
class Athlete:
    def __init__(self):
        #Initialization code…
```

- **The target identifer is assigned to the self argument.** Without it, the Python interpreter can't work out which object instance to apply the method invocation to. The self argument helps identify

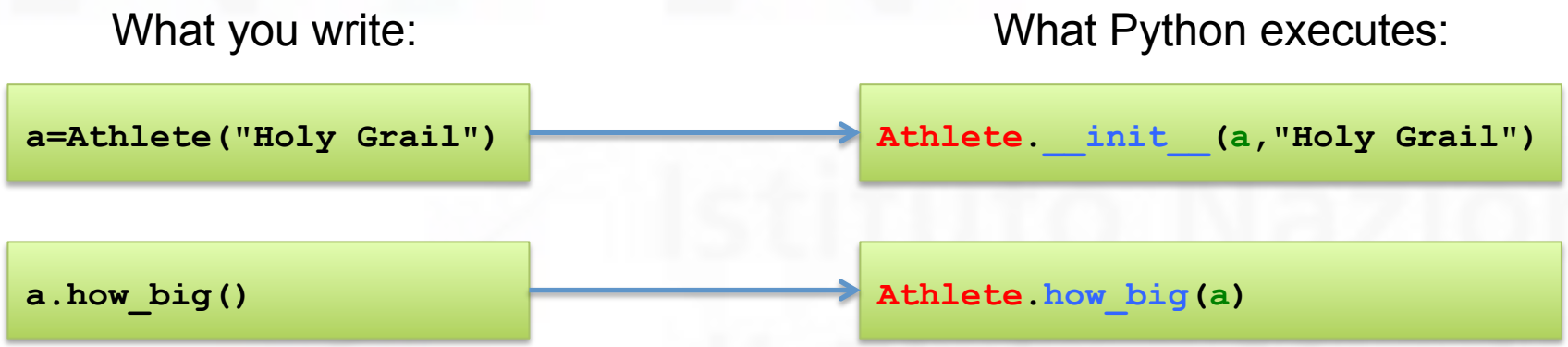
Every method's first argument is `self`

- In fact, not only does the `__init__()` method require self as its first argument, but *so does every other method defined within your class*.
- Python arranges for the first argument of every method to be the invoking (or *calling*) object instance.
- Let's extend the sample class to store a value in a object attribute called `thing` with the value set during initialization. Another method, called `how_big()`, returns the length of thing due to the use of the `len()` BIF:

```
class Athlete:
    def __init__(self, value=0):
        self.thing = value
    def how_big(self):
        return len(self.thing))
```

The `init` code now assigns a supplied values to a class attribute called `self.thing`, as usual not defined previously.
Note the use of `self` to identify the calling object istance.

When you invoke a class method on an object instance, Python arranges for the first argument to be the invoking object instance, which is *always* assigned to each method's self argument. This fact alone explains why self is so important and also why self needs to be the *first argument to every object method you write*:



Now go back to our last code (Dict3.py). The `sanitize()` function can remain unaltered, but now you want the function `get_run_data()` to return an instance of the Athlete class, that you have to define.

The class must have a method `top3()` that returns the three shorter times recorded.

First of all define the new class (use IDLE):

- What are its attributes?
- What are its methods?

```
class Athlete:
    def __init__(sel
        self.name=a_
        self.dob=a_d
        if a_times==
            a_times=
        self.times=a
    def top3(self):
        return sorts
```

```
>>> class prova:
        def __init__(self,lista=[]):
            self.mialista=lista

>>> a=prova()
>>> b=prova()
>>> a.mialista.append('ciccio')
>>> a.mialista
['ciccio']
>>> b.mialista
['ciccio']
```

```
s=None):
```

use a mutable object as
alue: it is initialized only on
stance! Try this:

```
elf.times]))[0:3]
```

You can modify the `get_run_data()` function and write the code that improve Dict3.py

```
def sanitize(time_string)       #unchanged
class Athlete                   #just defined

def get_run_data(filename):
    try:
        with open(filename) as f:
            data=f.readline()
        templ=data.strip().split(',')
        return Athlete(templ.pop(0),templ.pop(0),templ)
    except IOError as ioerr:
        print "File Error: %s" % ioerr
        return None

anna=get_run_data('anna2.txt')
print anna.name + "'s Fastest Times are: " + str(anna.top3())
```

http://bit.do/Class1-py

The function creates and returns a Athlete class object

Use the dot notation to get at your data and functions

Now you can simply add methods to **encapsulate** new functionality you need within your class.

Let's add two methods to your class. The first, called `add_time()`, appends a single additional timing value to an athlete's timing data. The second, `add_times()`, extends an athlete's timing data with one of more timing values supplied as a list.

```
def add_time(self, time_value):
    self.times.append(time_value)
def add_times(self, list_of_times):
    self.times.extend(list_of_times)
```

http://bit.do/Class2-py

You've **packaged** your code with your data and created a custom class from which you can create objects that share behaviors. And when extra functionality is required, **add more methods** to implement the required functionality.

By **encapsulating** your athlete code and data within a custom class, you've created a much more **maintainable** piece of software. By defining your own API with `add_time()` and `add_ times()`, you leave open the possibility that the way the data is stored within your class can change in the future (obviously, only if such a change makes sense).

It is worth noting that one of the reasons for using object orientation is to hide away the details of a class's implementation from the users of that class.

But … isn't your Athlete class wasteful? Your Athlete class does indeed behave like a **list** *most of the time*, and you've added methods to expose some list functionality to the users of your class. **Then it's true: *you are* reinventing the wheel here**. Athlete class only differs from Python's list due to the inclusion of the name and dob object attributes.

Inheritance

- You can use a basic concept of obect oriented programming, the inheritance, and build a "new style class", that are a **subclass** that inherit from the built-in classes.
- A subclass **inherits** from the class it derives from the methods and the attributes, and can specialize the parent class with new methods and attributes, or can rewrite the parent's methods.
- You can subclass also a custom defined class.
- Be warned though: Python does not care about interfaces, so you do not need to define interface classes and implementation classes.
- Python is a **duck typed** language.

Duck typing

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck. (Attributed to James Whitcomb Riley)

- In a duck typed language all that matters is the interface of the used object, and not its type.
- In a non-duck-typed language you can create a function that takes as input an object of "duck" type, and calls the `walk()` and `quack()` functions. **The language checks that the object has the correct type (and exposes the correct interface)**
- In a duck-typed language an equivalent function takes as input an object of ANY type and tries to call `walk()` and `quack()`. If the methods are not present the language raises a runtime exception, if the object has them the methods are executed.
- **Polymorphism without inheritance**: it is not necessary that classes inherit from a common interface, because the function does not require it!

Duck typing

```
>>>class Duck:
    def quack(self):
        print "Quaaack"
    def feathers(self):
        print "The duck has white and gray feathers"
>>>class Person:
    def quack(self):
        print "The person imitates a duck."
    def feathers(self):
        print "The person takes a feather from the ground and shows it."
    def name(self):
        print "John Smith"

>>>def in_the_forest(obj):
    obj.quack()
    obj.feathers()

>>>def game():
    donald = Duck()
    john = Person()
    in_the_forest(donald)
    in_the_forest(john)

>>>game()
```

```
>>> class NamedList(list):
        def __init__(self, a_name):
            list.__init__([])
            self.name=a_name

>>> johnny=NamedList("John Paul Jones")
>>> type(johnny)
<class '__main__.NamedList'>
>>> dir(johnny)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__delslice__', '__dict__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getslice__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__',
'__lt__', '__module__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__setslice__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'append', 'count', 'extend', 'index', 'insert', 'name',
'pop', 'remove', 'reverse', 'sort']


>>> johnny.append("Bass Player")
>>> johnny.extend(['Composer','Arranger','Musician'])
>>> johnny
['Bass Player', 'Composer', 'Arranger', 'Musician']
>>> johnny.name
'John Paul Jones'
>>> for attr in johnny:
        print johnny.name + " is a " + attr +"."
```

Provide the name of the class that this new class derives from.

Initialize the derived from class, and then assign the argument to the attribute.

“johnny” is a “NamedList”

“johnny” can do everything a list can, as well as store data in the “name” attribute.

Built-in list methods

“johnny” is like any other list, so feel free to use it wherever you’d use a list.

```
class Athlete:
    def __init__(self, a_name, a_dob=None, a_times=None):
        self.name=a_name
        self.dob=a_dob
        if a_times==None:
            a_times=[]
        self.times=a_times
    def top3(self):
        return sorted(set([sanitize(t) for t in self.times]))[0:3]
    def add_time(self, time_value):
        self.times.append(time_value)
    def add_times(self, list_of_times):
        self.times.extend(list_of_times)
```

Let's rewrite this class to inherit from the built-in list class.

```
class AthleteList(list):
    def __init__(self, a_name, a_dob=None, a_times=None):
        if a_times==None:
            a_times=[]
        list.__init__([])
        self.name=a_name
        self.dob=a_dob
        self.extend(a_times)
    def top3(self):
        return sorted(set([sanitize(t) for t in self]))[0:3]
```

Inherit from the built-in list class.

Initialize the derived from class and then assign the argument to the attribute.

The data itself is the timing data, so the "times" attribute is gone.

Let's replace the Athlete class code with our new AthleteList class code. Don't forget to change `get_run_data()` to return an AthleteList object instance.

http://bit.do/Class3-py

Class attributes

- A class can have two types of attributes:
  1. *Data attributes*: are the variables that identify a particular object instance of the class. Each instance has its own value for them.
  2. *Class attributes*: are relative to the entire class. All instances of the class share the **same value** for them. In other languages they are called "static data member".

- Class attributes are defined within the definition of a class, but outside of any method

- Since there is only one attribute per class and not one for instance, you can access them or by the istance name, or in the class methods by `self.__class__.name`

```
>>> class point:
      counter=0
      def __init__(self,x,y):
        self.x=x; self.y=y
        point.counter += 1

>>> for i in range(1000):
        p = point(i*0.01, i*0.001)
>>> point.counter
1000
>>> p.counter
1000
>>> p.__class__.counter
1000
```

This is the class attribute

Access to the class attribute by the class name

Three different methods to read the class attribute

If you assign a value to counter **THROUGH an instance**, actually you create a new attribute! **The local attribute hides the class attribute.**

```
>>> for i in range(1000):
        p = point(i*0.01, i*0.001)
>>> p.counter=0
>>> point.counter
1000
>>> p.counter
0
>>> p = point(0,0)
>>> p.counter
2001
```

This is a NEW attribute!

The class attribute is not changed

Now you cannot read the class attribute from p: you get the istance one!

From other istances nothing changes

Built-in methods and attributes

- The following attributes exist for any class:
  1. `__doc__`: is the variable that keeps the documentation string for the class (written under the keyword class)
  2. `__class__`: is the variable that provides a reference to the class from any instance of it
  3. `__module__` : is the variable that provides a reference to the module in which the particular class is defined

- Classes contain a few methods included by default eve if they are not explicitly defined. For example `__init__` or `__repr__` exist for all classes.

- Many of these methods are automatically invoked when built-in operators are used. For example `print f`, or simply type f on the Python command prompt, calls `f.__repr__()` , that if not overwritten produces a tring representation of the f object.

```
>>> p
<__main__.point instance at 0x10a9b7560>
```

- All built-in methods have a double underscore around the name, and can always be overridden by the programmer
  1. `__init__`: it's the class constructor
  2. `__len__`: defines how `len(obj)` works on class istances

Copy

- As we have seen you can copy an immutable objects with an assignment, a list with a slice, a dictionary with the `copy()` function. An assignment ALWAYS make a copy of reference only. So how do you copy a class istance?
- You have to use the `copy` module, with the functions `copy.copy()` and `copy.deepcopy()`. The first method does a shallow copy, that is nested mutable objects are not copied (dict.copy() and list slicing are shallow).

```
>>> import copy
>>> newObj = copy.copy(myObj)          # shallow copy
>>> newObj2 = copy.deepcopy(myObj2)    # deep copy
```

- In order for a class to define its own copy implementation, it can define special methods `__copy__()` and `__deepcopy__()`. The former is called to implement the shallow copy operation; no additional arguments are passed. The latter is called to implement the deep copy operation; it is passed one argument, the memo dictionary. If the __deepcopy__() implementation needs to make a deep copy of a component, it should call the deepcopy() function with the component as first argument and the memo dictionary as second argument.

Comparison

- Instead of a single method to address sorting and all six comparison operators, there's a one-to-one correspondence between operator and method:

```
1.  ==  __eq__()
2.  !=  __ne__()
3.  >   __gt__()
4.  >=  __ge__()
5.  <   __lt__()
6.  <=  __le__()
```

- In this way you can manage partially ordered sets, where some pairs (A, B) simply don't have an order relative to each other. For example, if your abstraction for software packages includes the hardware architecture, then you lose the nice clear total ordering that you have with a totally ordered set like integers. You gain something extremely valuable: a more accurate abstraction.

__call__

```
>>> class F:
    def __init__(self, a=1, b=1, c=1):
        self.a = a; self.b = b; self.c = c
    def __call__(self, x, y):
        return self.a + self.b*x + self.c*y*y

>>> f = F(a=2, b=4)
>>> v = f(2, 1) + f(1.2, 0)
>>> v
17.8
```

When you call `f(1.2,0)` Python runs

```
f.__call__(1.2,0)
```

This feature is useful for implementing parametric functions, where it is necessary to distinguish between parameters and independent variables. The previous class implements the function

$f(x,y;a,b,c) = a+bx+cy^2$

x and y are the independent variables. a,b,c are istance attributes.
F is essentially a function factory.