The cell above loads the visual style of the notebook when run.

In [1]:
```
from IPython.core.display import HTML
css_file = '../styles.css'
HTML(open(css_file, "r").read())
```

Out[1]:

Storing multiple values in lists

Learning Objectives

- Explain what a list is.
- Create and index lists of simple values.

Just as a `for` loop is a way to do operations many times, a list is a way to store many values. Unlike NumPy arrays, lists are built into the language (so we don't have to load a library to use them).

We create a list by putting values inside square brackets:

In [2]:
```
odds = [1,3,5,7]
print ('Odd numbers are:',odds)
```

```
Odd numbers are: [1, 3, 5, 7]
```

We select individual elements from lists by indexing them:

In [3]:
```
print ('first and last:', odds[0], odds[-1])
```

```
first and last: 1 7
```

and if we loop over a list, the loop variable is assigned elements one at a time:

In [4]:
```
for number in odds:
    print(number)
```

```
1
3
5
7
```

There is one important difference between lists and strings: we can change the values in a list, but we cannot change the characters in a string.

For example:

In [5]:
```
names = ['Newton', 'Darwing', 'Turing'] # typo in Darwin's name
print ('names is originally:', names)
names[1] = 'Darwin' # correct the name
print ('final value of names:', names)
```

```
names is originally: ['Newton', 'Darwing', 'Turing']
final value of names: ['Newton', 'Darwin', 'Turing']
```

works, but:

In [6]:
```
name = 'Bell'
name[0] = 'b'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-dce57b136c2e> in <module>
      1 name = 'Bell'
----> 2 name[0] = 'b'

TypeError: 'str' object does not support item assignment
```

doesn't!

Changes

Data which can be modified in place is called mutable (reference.html#mutable), while data which cannot be modified is called immutable (reference.html#immutable). Strings and numbers are immutable. This does not mean that variables with string or number values are constants, but when we want to change the value of a string or number variable, we can only replace the old value with a completely new value. For example, consider the diagram below, which shows what happens when we change the value of an immutable variable.

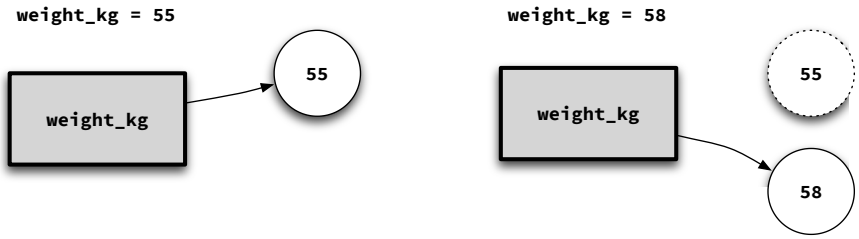

Figure: Changing immutable objects

Lists and arrays, on the other hand, are mutable: we can modify them after they have been created. We can change individual elements, append new elements, or reorder the whole list. For some operations, like sorting, we can choose whether to use a function that modifies the data *in place* or a function that returns a modified copy and leaves the original unchanged. Consider the diagram below, which illustrates changing a list in place.

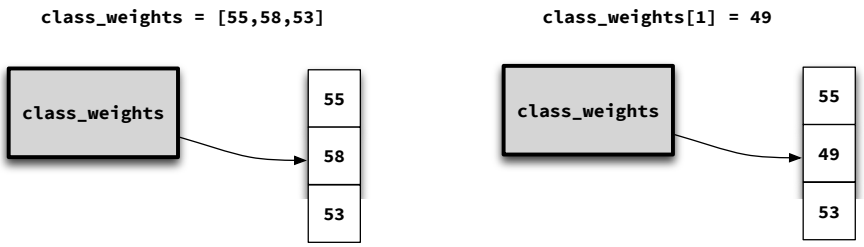

Figure: In-place change to mutable object

Check your understanding

Look at the code below. What do you expect it to do? Check your answer with the person next to you.

```
ages = [20,23,18,30]
new_ages = ages
new_ages[1] = 64
print(ages)
```

```

Now run the code in the cell below. Does it do what you expect?

The code should take the second element of the list ages and replace it with the new input, 64, while the rest of the list remains unchanged

In [7]:
```
ages = [20,23,18,30]
new_ages = ages
new_ages[1] = 64
print(ages)
```

```
[20, 64, 18, 30]
```

Mutability and copies

In Python, the statement `new_ages = ages` doesn't make a **copy** of `ages`. Instead, it adds a new label to the same bit of computer memory. You can think of this like putting two sticky labels on the same box. This has important consequences for mutable objects like lists.

In [ ]:
```
from IPython.display import HTML
HTML('<iframe width="700" height="320" frameborder="0" src="http://pythontutor.
```

When you modify `new_ages`, you're changing the memory that `ages` is looking at. If you want variables with mutable values to be independent, you must make a copy of the value when you assign it. You can use the built-in `list()` command to create a new list, so we do not modify a list we did not mean to
```

```
    new_ages = list(ages)
```

```

Because of pitfalls like this, code which modifies data in place can be very difficult to understand! If you find yourself having trouble with variables changing when you don't expect, or vice-versa, you've likely fallen foul of this subtlety. It can be very useful to try out your code at pythontutor.com (http://pythontutor.com) - this will produce diagrams like the ones above to help you visualise how variables are being assigned and changed. If you are interested in understanding this concept more fully, I can strongly recommend the excellent talk here (http://nedbatchelder.com/text/names1.html).

There are many ways to change the contents of lists besides assigning new values to individual elements. Let's look at some of them briefly:

In [ ]:
```
odds = [1,3,5,7]
odds.append(11)
print ('Odd numbers after adding a value:', odds)
```

In [ ]:
```
del odds[0]
print ('Odd numbers after removing the first element:', odds)
```

In [ ]:
```
odds.reverse()
print ('Odd numbers after reversing:', odds)
```

## 📌 Unpacking from lists

Python has a really neat syntax for getting values back out of a list. For example, the following code unpacks values from a list of planet masses
```

In [9]:
```
# planet masses in units of Earth's mass
planet_masses = [0.0553, 0.815, 0.107, 317.8, 95.2]

mercury, venus, mars, jupiter, saturn = planet_masses

print ("Saturn's mass is ", saturn, " Earth masses")
```

```
Saturn's mass is  95.2  Earth masses
```

Turn a string into a list

Use a for-loop to convert the string "hello" into a list of letters:

```
["h", "e", "l", "l", "o"]
```

>Hint: You can create an empty list like this:

>```python
my_list = []

In [10]:
```
# INSERT YOUR CODE HERE

hello = 'hello'

list = []

for i in 'hello':
    list.append(i)

print(list)
```

```
['h', 'e', 'l', 'l', 'o']
```

In [ ]: