Working with modules and functions

```
>>> import utils
>>> reload(utils)
<module 'utils' from 'utils.pyc'>
```

`reload()` reloads an *already loaded* module from the file containing the module.

```
>>> dir(utils)
['__builtins__', '__doc__', '__file__', '__name__', 'dict2file', 'file2dict',
'find_root', 'greet', 'print_and_return', 'print_dict', 'reverse']
```

`dir()` displays all the *names* defined within a module (or indeed in any type of object).

```
>>> callable(utils.file2dict)
True
>>> callable(utils.__doc__)
False
```

`callable()` tells us whether or not we can call something.

9

We already know how to load a module in Python using the `import` statement. We've also seen that if we make changes to the module we need to reload it by using the `reload()` function. If we try to import the module again, Python will not do anything since it knows it has already loaded (imported) the module. We have to explicitly tell it to `reload()` it.

How can we find out what functions are defined in a module? This is unfortunately not straightforward, although we can easily find out all the *names* that are defined in the module using the `dir()` function. These names will not be just the functions defined in the module though, they will be a mixture of any variables defined in the module, any functions defined in the module and also some special things created by Python (such as `__doc__` which contains the module's doc string). The special things created by Python will always be called something like *__name__*, i.e. they will be prefixed and followed by two underscore (_) characters. In general you disregard these, apart from the doc string (`__doc__`) which should contain useful information about the module.

Note that we can use the `dir()` function not just on modules, but on *any* object and it will tell us all the names that are defined within that object. (In case you were wondering, *everything* in Python is an object: modules, functions, variables, everything. What do we mean by "object" here? Basically it's a programming jargon term for a special sort of structure that can have both variables and functions defined within it.)

So how *can* we tell whether one of those names is a function or not? Well, we could try using the name as a function and seeing what happened, but that would quickly get tedious (as well as possibly giving false negatives). There's a better way: use the `callable()` function. The `callable()` function tells us whether a given name is callable, i.e. whether we can call it, i.e. if we can use it as a function. (However, you should be aware that there are pathological circumstances in which the `callable()` function will tell us that something is callable even when a call to it would fail; however, the converse (telling us something isn't callable when it is) should never happen.)

Augmented assignment

```
>>> a = 1
>>> a += 1
>>> a
2

>>> a -= 1
>>> a
1

>>> a *= 4
>>> a
4
```

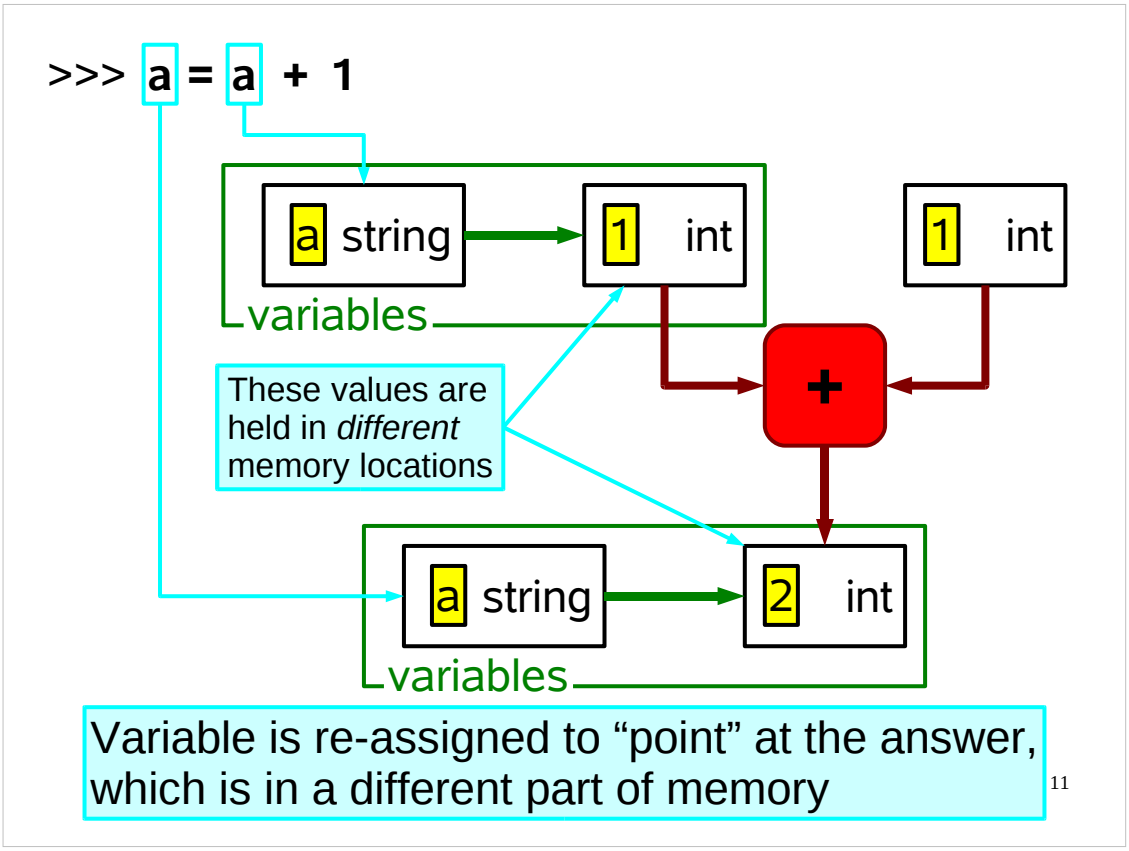
```
>>> a = 1
>>> a = a + 1
>>> a
2

>>> a = a - 1
>>> a
1

>>> a = a * 4
>>> a
4
```

Similarly, we can also use the following for…

division:	/=
exponentiation:	**=
remainder:	%=

10

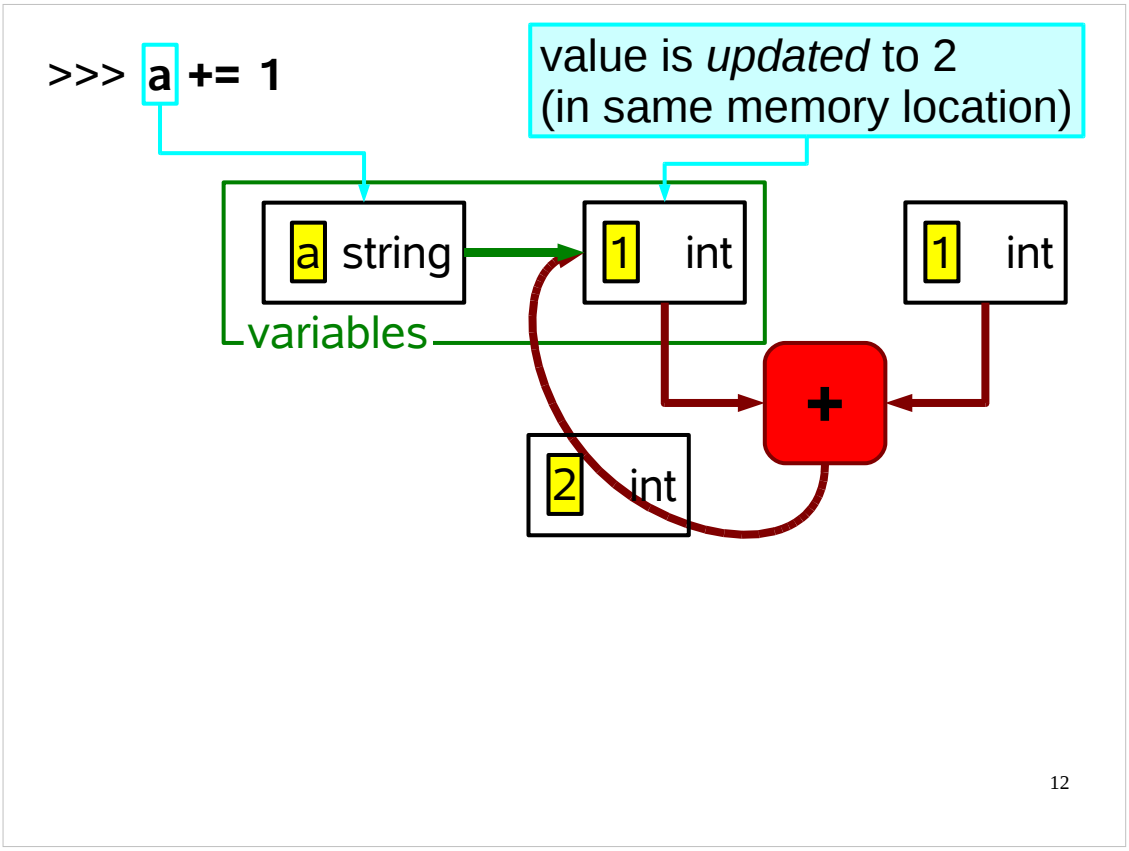
When we use the forms `+=`, `-=`, `*=`, `/=`, `**=` and `%=` we are doing what is known as *augmented assignment*. Basically, this is a combination of an operation (`+`, `-`, `*`, `/`, `**` or `%` respectively) and an assignment (assigning the result of that operation to a variable). You can also think of it as "assignment in place" because Python will attempt to update the variable's value rather than creating a temporary value and then "pointing" the variable at that new value (which is what it does when we give it something like `a = a + 1`).



When you tell Python to do something like:

```
a = a + 1
```

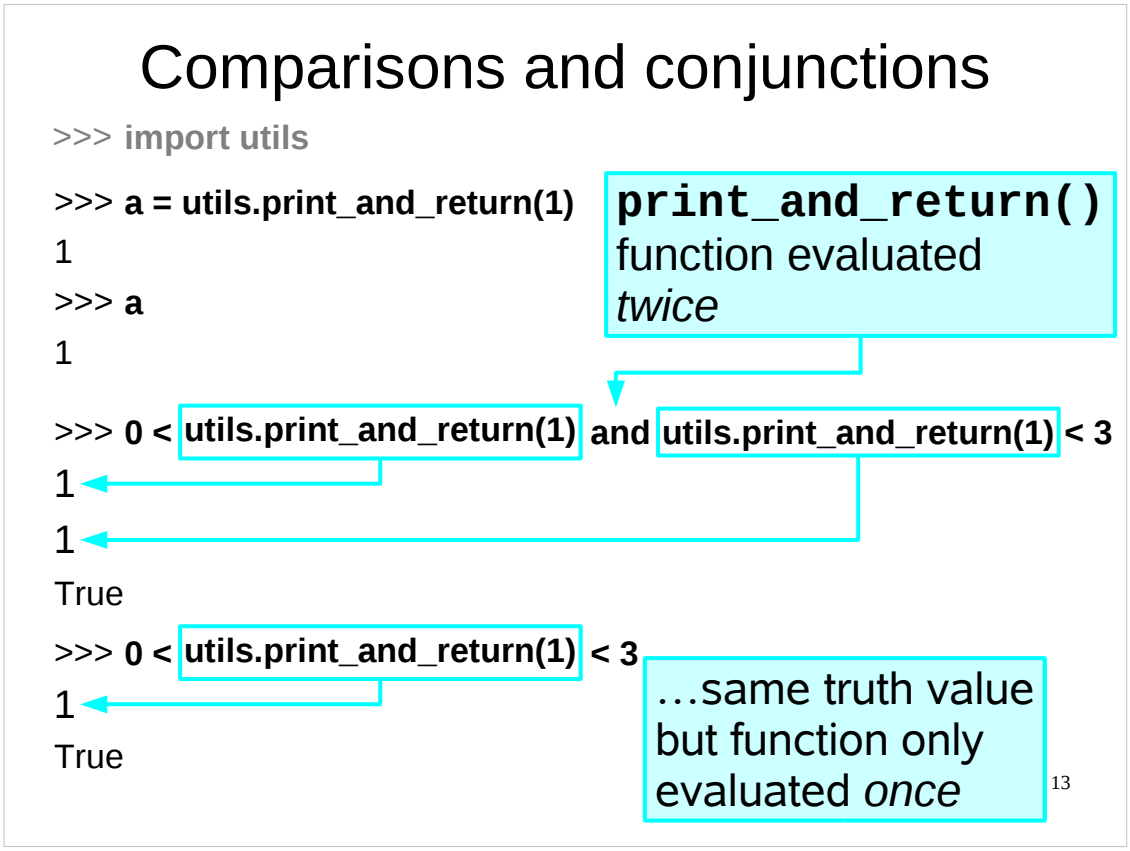
what it does is look up the value of `a`, then adds `1` to that value and stores the answer in a *different* memory location. It then updates `a` to “point” to that new memory location and releases the memory that stored the previous value of `a`.



However, when you tell Python to do an augmented assignment, such as:

```
a += 1
```

what it does is look up the value of `a`, then adds `1` to that value and stores the answer in the *same* memory location (if it can), i.e. it updates `a` "in place".



We've already met the `and` conjunction for joining two comparisons together. However, there is a more compact way of doing something similar for the special case where we are doing something like:

a *compare* b `and` b *compare* c

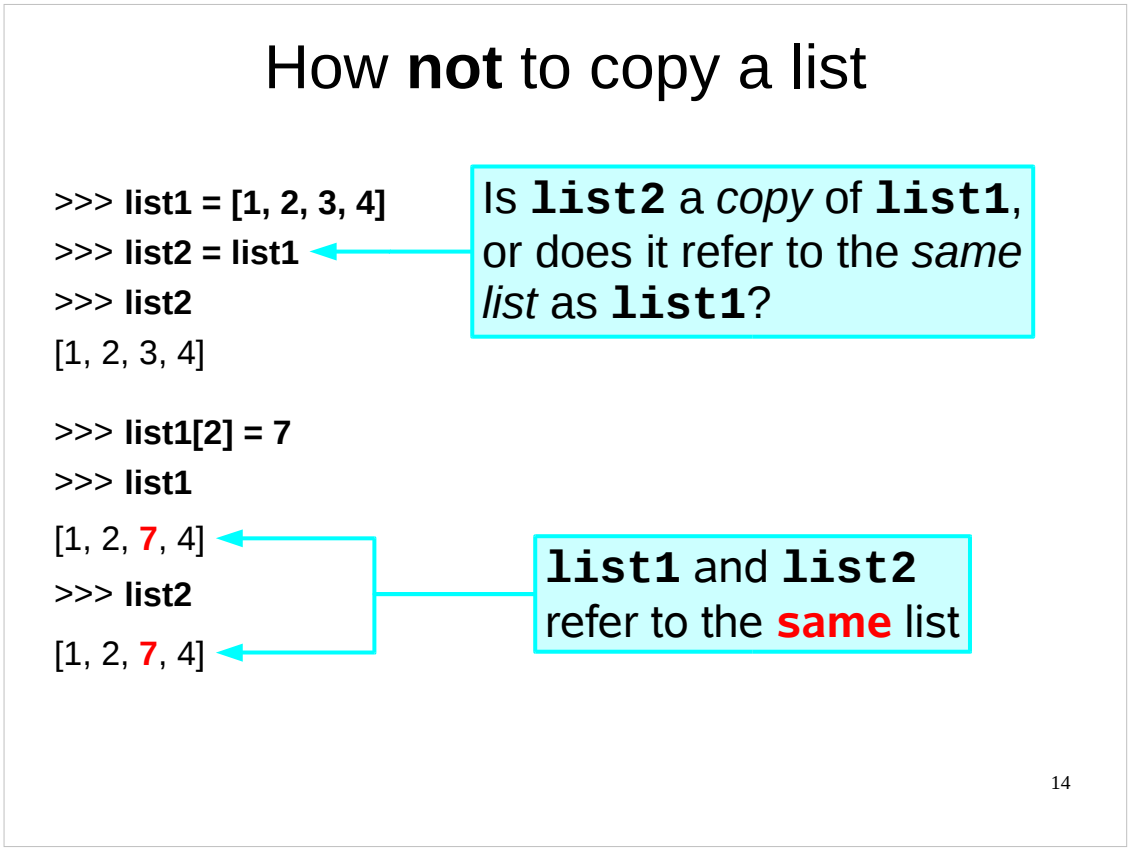
(where "compare" stands for any comparison operator, such as "<" or ">"; note that the comparison operators used to compare a to b and b to c ***do not*** have to be the same). In this particular case, we can just drop the "`and`", thus:

a *compare* b *compare* c

e.g. `a < b < c`, or even `a < b >= c`.

However, there is one important thing to note: in this more compact form, b is only evaluated ***once***, whilst in "a *compare* b `and` b *compare* c", b may be evaluated ***twice***. We can easily see this if b is a function that has some side-effect (such as printing something on the screen) as in the slide above.

(The `print_and_return()` function is not a standard Python function. It was specially created for this course to illustrate this particular point. You will find it in the `utils` module in your course home directory. It just `prints` whatever argument it has been given and then `returns` that argument.)



If we've assigned a list to a variable (say a variable called `list1`) and we want to make a copy of that list (and assign that copy to another variable, say a variable called `list2`), we might be tempted to do something like this:

```
list2 = list1
```

Unfortunately this does ***not*** work in the way we might expect!

What happens is that both `list1` and `list2` now refer to the same list in the computer's memory. Changing `list1` will affect `list2` (and vice-versa), since they are both actually the same list. When we "copy" a list like this, we don't actually copy it at all, we just create a new variable that "points" to the same list that we had before. (This is sometimes called a "shallow copy".)

We can see that this is the case if we use the `id()` function. This function returns a constant, unique reference (an "identity") for each *unique* object that has been created. If two variables refer to the same object, then the `id()` function will return the same reference for both variables. (The reference will be an integer or long integer – what the `id()` function actually returns is the memory address at which the object is stored.) If you've typed in the Python on the slide above, you can try this function on `list1` and see what it returns:

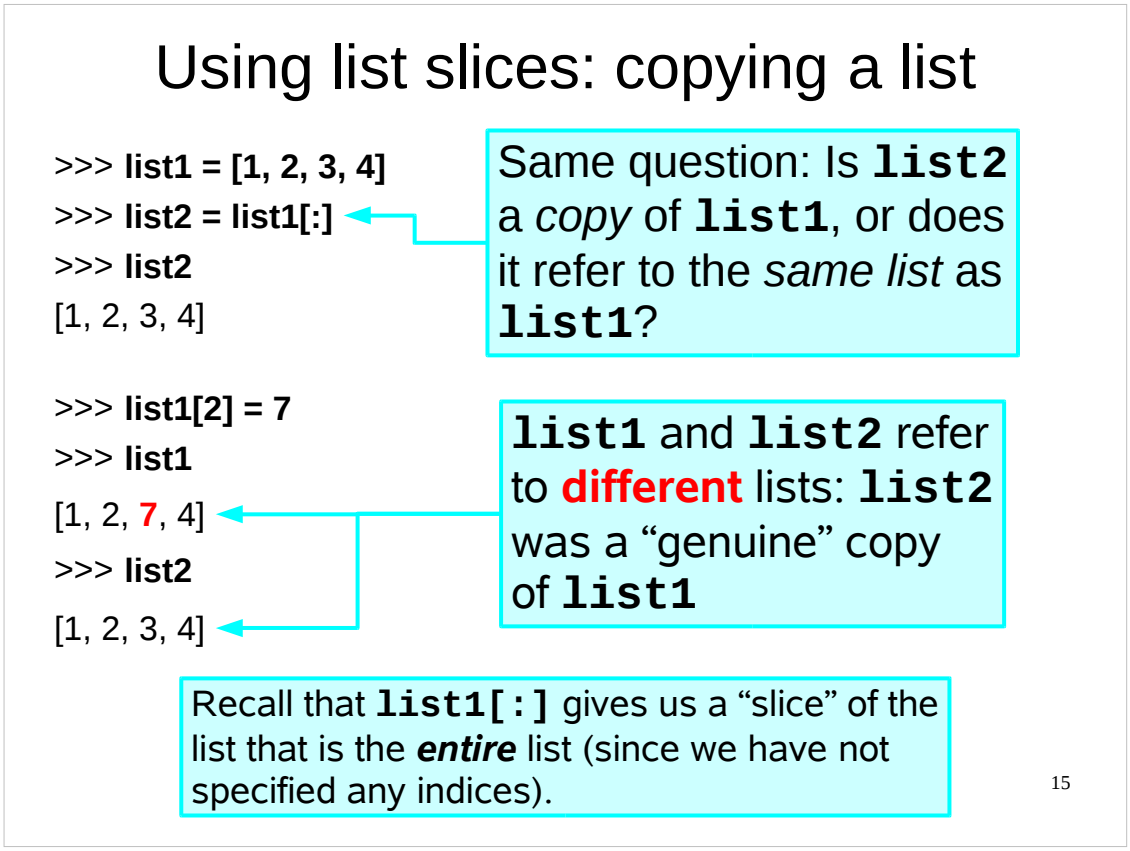
```
>>> id(list1)
```

and then on `list2`:

```
>>> id(list2)
```

You should find that `id()` returns the same value for both these variables (whatever that value might happen to be).

So how *can* we make a real copy of a list?…



Recall how we can get sections of a list: list *slices*. If `list1` is a list, we can get a "slice" of it using the syntax `list1[i:j]`, where `i` and `j` are indices of the list. `list1[i:j]` will give us all the items in the list from the item whose index is `i` up to and including the item whose index is `j-1`. We can exclude either or both of the indices in the slice; if we exclude both indices (so `list1[:]`) then the slice we get is the entire list.

In fact, that slice is a **copy** of the entire list. A real, genuine, honest-to-goodness copy that is a *different* list (with the same values in the same order), stored in a *different* memory location. (This is sometimes called a "deep copy".)

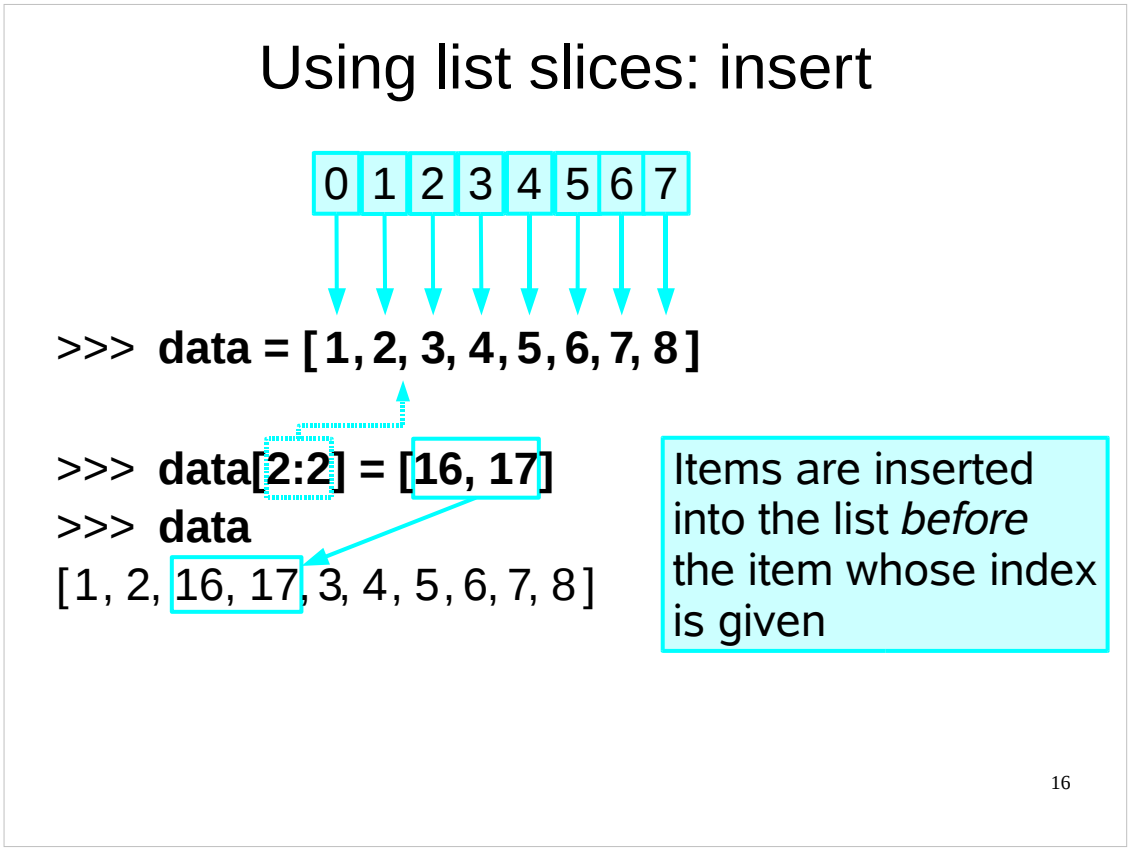
Again, we can see that this is the case using the `id()` function. If you've typed in the Python on the slide above, you can try this function on `list1` and see what it returns:

```
>>> id(list1)
```

and then on `list2`:

```
>>> id(list2)
```

You should find that `id()` returns *different* values for each variable (whatever those values might happen to be). That means that they refer to different objects in memory (which may or may not happen to have the same value).



Note that the items we are inserting have to come from a list, and we can insert as many (or as few) items as we like.

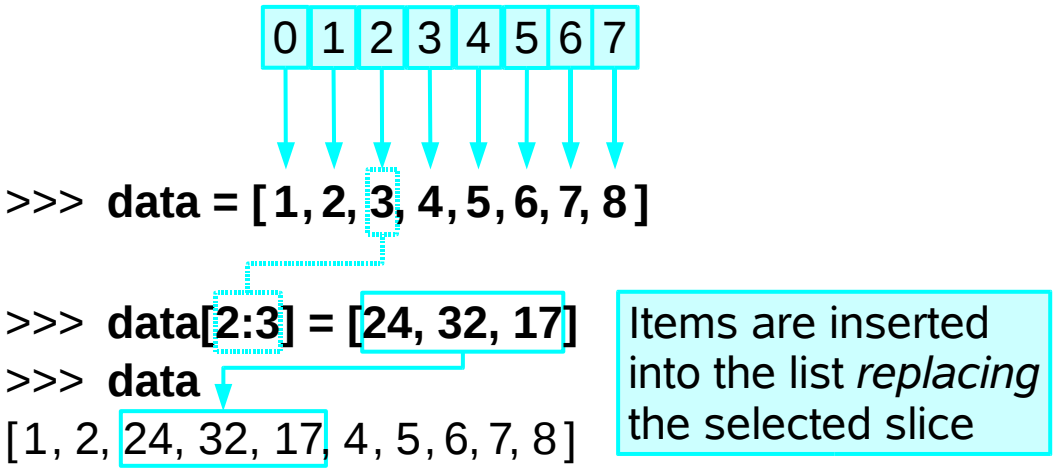
You may wonder why the insertion is *before* the given index rather than after it. Recall that a slice starts from the lower index and goes up to just before (one less than) the higher index. So the slice *i*:*i* at first glance seems nonsensical because it would have to start at item *i* and stop just *before* item *i*. So Python interprets this as being "the empty space just before item *i*", which does not actually contain a value, so that, for any list the slice *i*:*i* will evaluate to the empty list, e.g.

```
>>> data = [1, 2, 3, 4, 5, 6, 7, 8]
>>> data[2:2]
[]
```

You can also insert a *single* item into a list using the `insert()` method of the list, which inserts a single item **at** the given index, e.g.

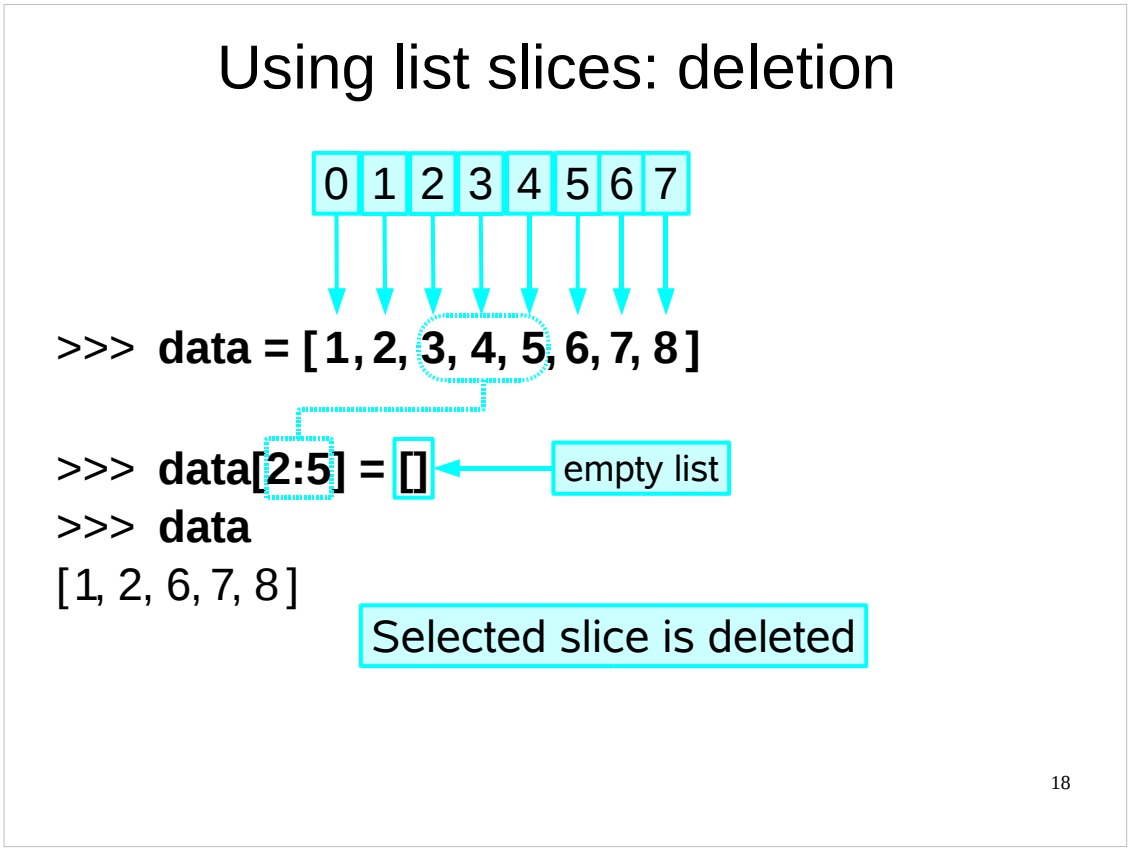
```
>>> data = [1, 2, 3, 4, 5, 6, 7, 8]
>>> data.insert(2,16)
>>> data
[1, 2, 16, 3, 4, 5, 6, 7, 8]
```

Using list slices: replace (and insert)



17

As mentioned earlier, the items we are inserting have to come from a list, and we can replace the slice with as many (or as few) items as we like. Thus, if the slice we're replacing is not the empty list (`[]`), as it was in the previous example, then we will actually be "inserting and replacing" rather than just inserting…



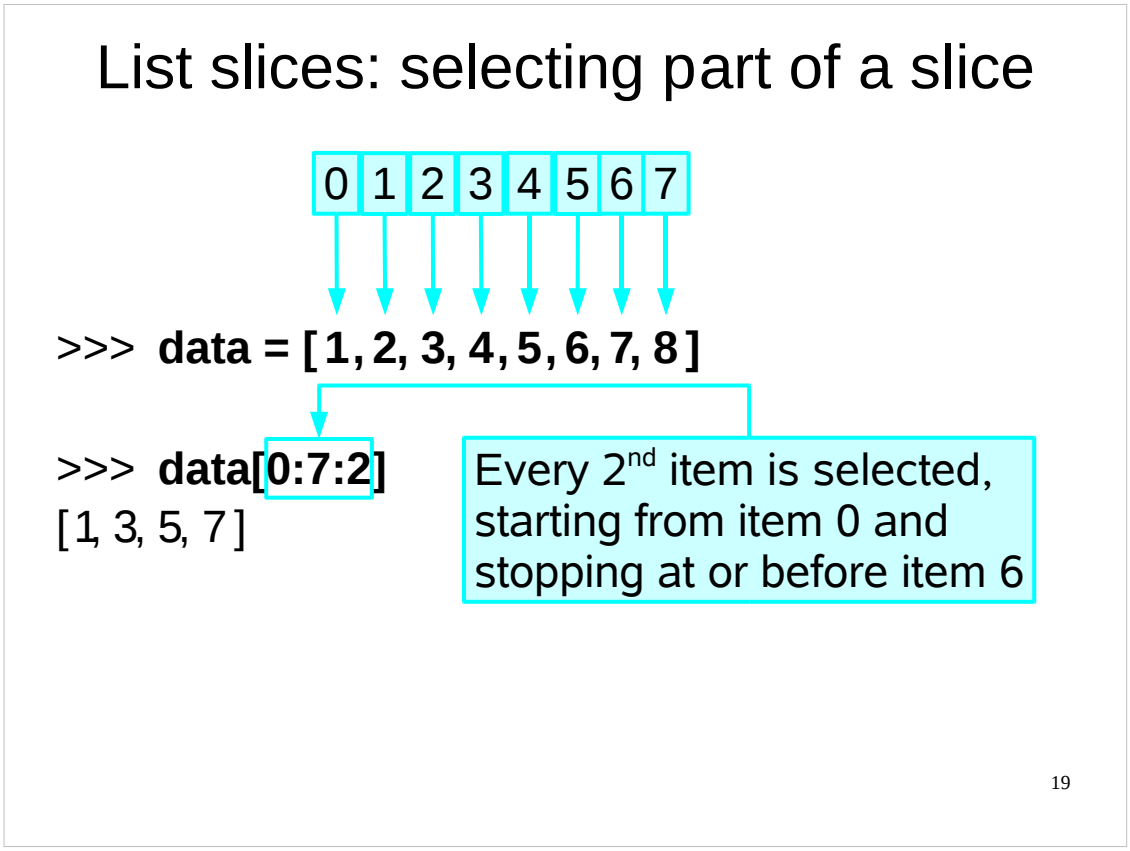
…which means that if we replace the slice with no items, i.e. the empty list (`[]`), then we'll actually delete the slice.

For comparison, remember that you can also delete a single item from a list using the list's `remove()` method. This method removes the *first* matching item in a list (wherever that item might be), e.g.

```
>>> data = [1, 2, 3, 2, 8]
>>> data.remove(2)
>>> data
[1, 3, 2, 8]
```

There's also another way you can delete items from a list: using the `del` operator. This operator can either delete a single item from a list, or an entire slice. `del list[i]` removes the item whose index is `i` from the list, whilst `del list[i:j]` removes the slice `i:j` from the list, e.g.

```
>>> data = [1, 2, 3, 2, 8]
>>> del data[2]
>>> data
[1, 2, 2, 8]
>>> del data[0:2]
[2, 8]
```



This may seem slightly odd until you get used to it. The way to think of it is that the slice *i*:*j*:*k* (which you can read as "the slice *i*:*j* in steps of size *k*") gives you the following items from the slice *i*:*j* –

item *i*
item *i* + *k*
item *i* + *2*k*
item *i* + *3*k*
item *i* + *4*k*

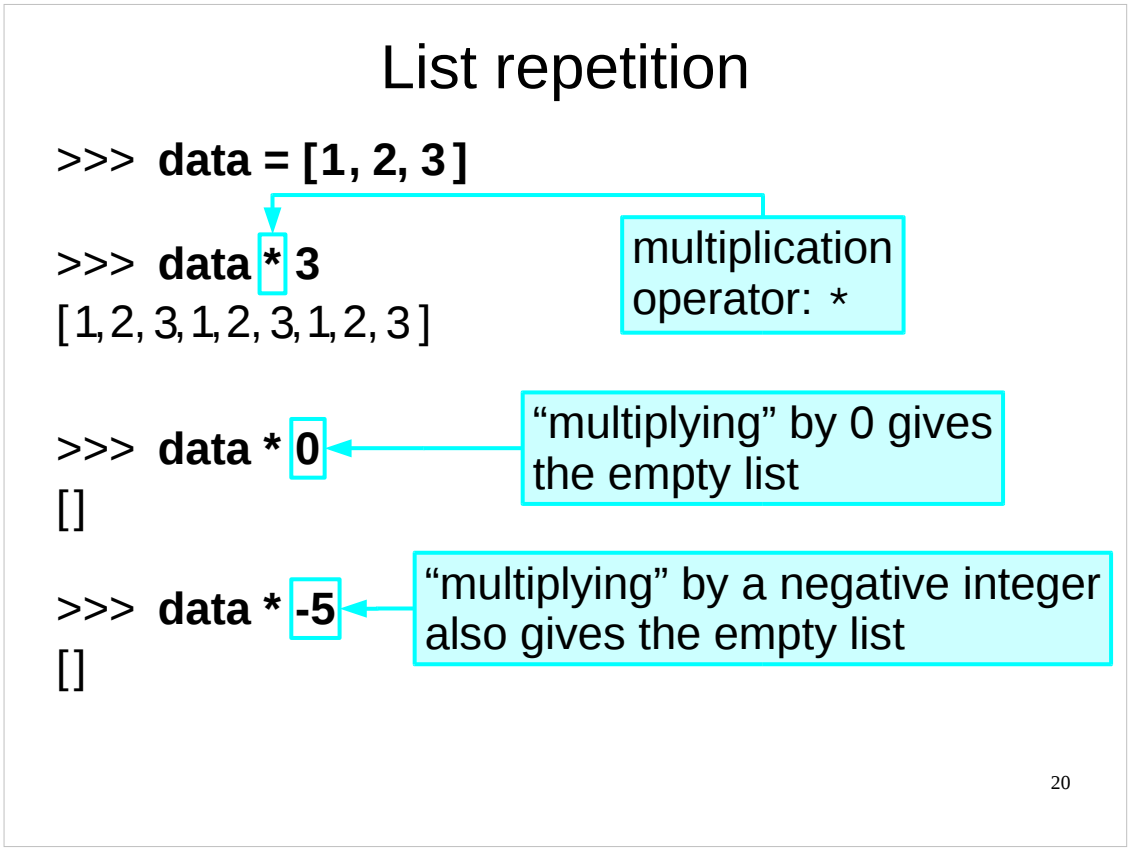
…and so on, up to (but **not** including) item *j*, i.e. (for the mathematically inclined) we stop at *i* + *n*k*, where

$$i + n*k < j \leq i + (n+1)*k$$

Having selected part of a slice in this way, you can replace the items you've selected in a similar manner to the way in which we've seen we can replace an ordinary slice of a list, i.e. we set the selected part of the slice equal to another list of items. There is one restriction, though: we must replace this part of a slice with exactly the *same* number of items, e.g.

```
>>> data = [1, 2, 3, 4, 5, 6, 7, 8]
>>> data[0:7:2] = [3, 9, 15, 21]
>>> data
[3, 2, 9, 4, 15, 6, 21, 8]
```

This restriction means that we can't remove these sorts of parts of a slice by setting them equal to the empty list (`[]`), as we can with normal slices. Oh, well, you can't have everything.



If we "multiply" a list by an integer (either a normal integer or a long integer) we will get list *repetition*: a new list is generated which consists of the original list repeated the specified number of times. If we "multiply" a list by a negative integer or by zero, then we get the empty list (`[]`).

Note that we can't "multiply" a list by a floating point number or a complex number.

Evaluate the following Python statements in your head. What are the items in the list `primes` after each statement?

```
>>> primes = [2, 3, 5, 7, 11, 13, 17, 19]
>>> primes[0::3] = [1, 6, 16]
>>> del primes[2:5]
>>> primes[2:2] = [5] * 3
>>> primes[0::4] = [2, 11]
>>> primes[3::3] = [7, 17]
```

Now try them interactively in Python and see if you were correct.

21

If you run into problems with this exercise, or if you don't understand any of the Python on the slide above, please ask the course giver or a demonstrator for help.

When you've finished take a short break of a minute or two – that means **stop** staring at the computer screen and move around, relax, etc.

(Note: If you've done it correctly, you should find that the items in `primes` when you've finished are the same (and in the same order) as when you first assigned a list to `primes`.)

When **not** to use list repetition

```
>>> x = [ [0, 0] ] * 2
>>> x
[[0, 0], [0, 0]]
```

```
>>> x[0][0] = 1
>>> x
[[1, 0], [1, 0]]
```

probably ***not*** what we wanted to happen…

List repetition works fine if the list consists of simple data types (integers, floats, complexes, etc.) but with more complicated types (e.g. a list of *lists*) the new list contains "**shallow copies**" of the repeated item(s).

22

And now a very important "gotcha": list repetition, used in the wrong circumstances, will not behave the way we might expect.

If we use list repetition on a list of *lists*, then the new list consists of a set of "*shallow copies*" of the repeated items, as we see on the slide above. Thus, in the example above, instead of having a list of two items, each of which is a distinct list (that just happen to have the same values in the same order when we first set them up), we have a list of two items, each of which is the ***same*** list, c.f. what happened earlier when we tried to copy a list without using slices.

(Note that we get this wrong-headed behaviour whenever we use list repetition on a list whose items are themselves complicated types, such as lists or dictionaries.)

You may be wondering why we would want to have a list of lists like the one we want to create on the slide above. Such lists are often used as matrices. Since Python doesn't have a built-in matrix type, people often use a list of lists instead. So, on the slide above, `x` (if it behaved properly) could represent a 2×2 matrix. Similarly, the 4×4 identity matrix might be represented by the following list:

```
[[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]
```

If you are going to do serious matrix work in Python, though, you are better off using the NumPy module. This is not a standard Python module, but is freely available from:

http://www.scipy.org/Download

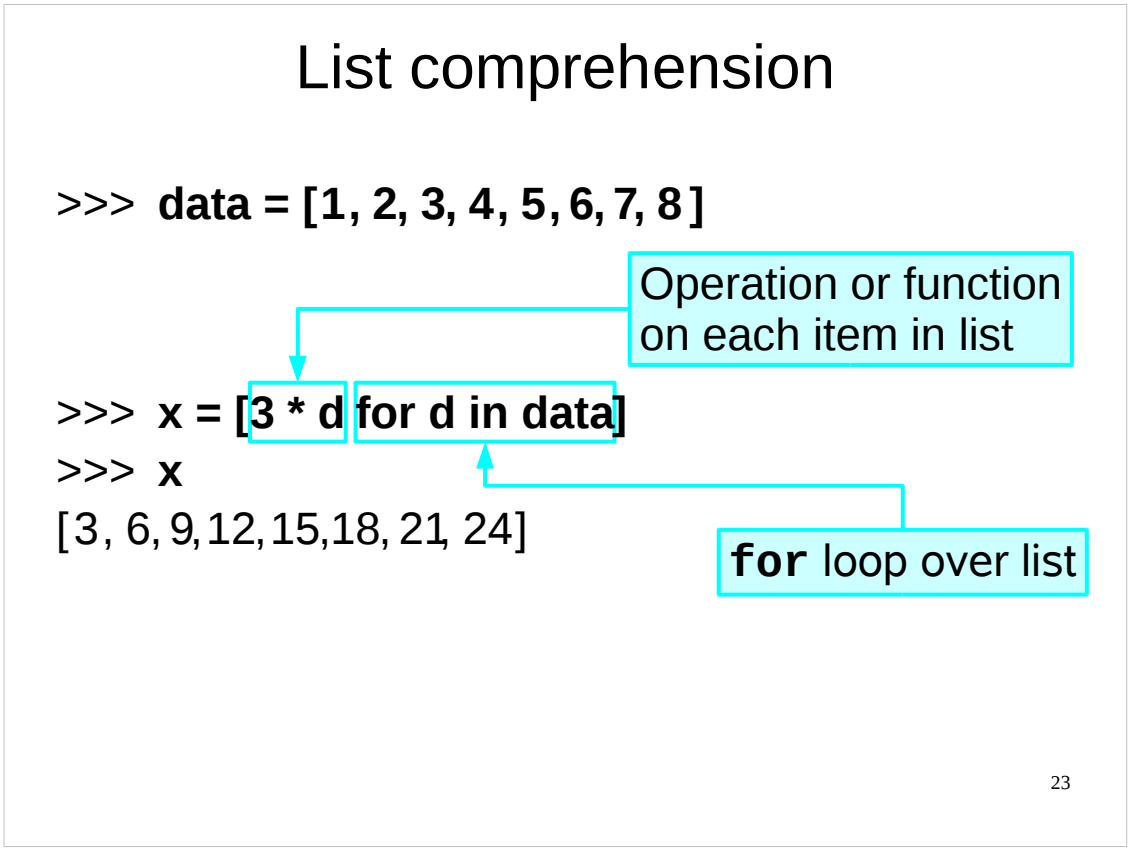
For documentation on the NumPy module, see:

http://docs.scipy.org/

Using matrices in Python and basic use of the NumPy module are covered in the "Python: Interoperation with Fortran" course. For details of this course, see:

http://www.training.cam.ac.uk/ucs/course/ucs-pythonfort

So how can we do something like list repetition for a list of lists? Well, first we need to know a little more Python…



The Python technique we need is called *list comprehension*.

List comprehension is a handy technique for quickly creating one list from another. Basically, you specify an operation or function to be carried out on each item in an existing list. Python will then construct a new list for you whose items are the results of carrying out the specified operation or function on the items (in order) in the old list.

Note that the old list doesn't, in fact, have to be a list at all: anything that you can legitimately treat as a list for the purposes of a `for` loop (a dictionary, a `file` object, etc) can be used.

List comprehensions to repeat a list

```
>>> x = [ [0, 0] for d in range(0,2) ]

>>> x
[[0, 0], [0, 0]]

>>> x[0][0] = 1
>>> x
[[1, 0], [0, 0]]
```

Yay! It works!

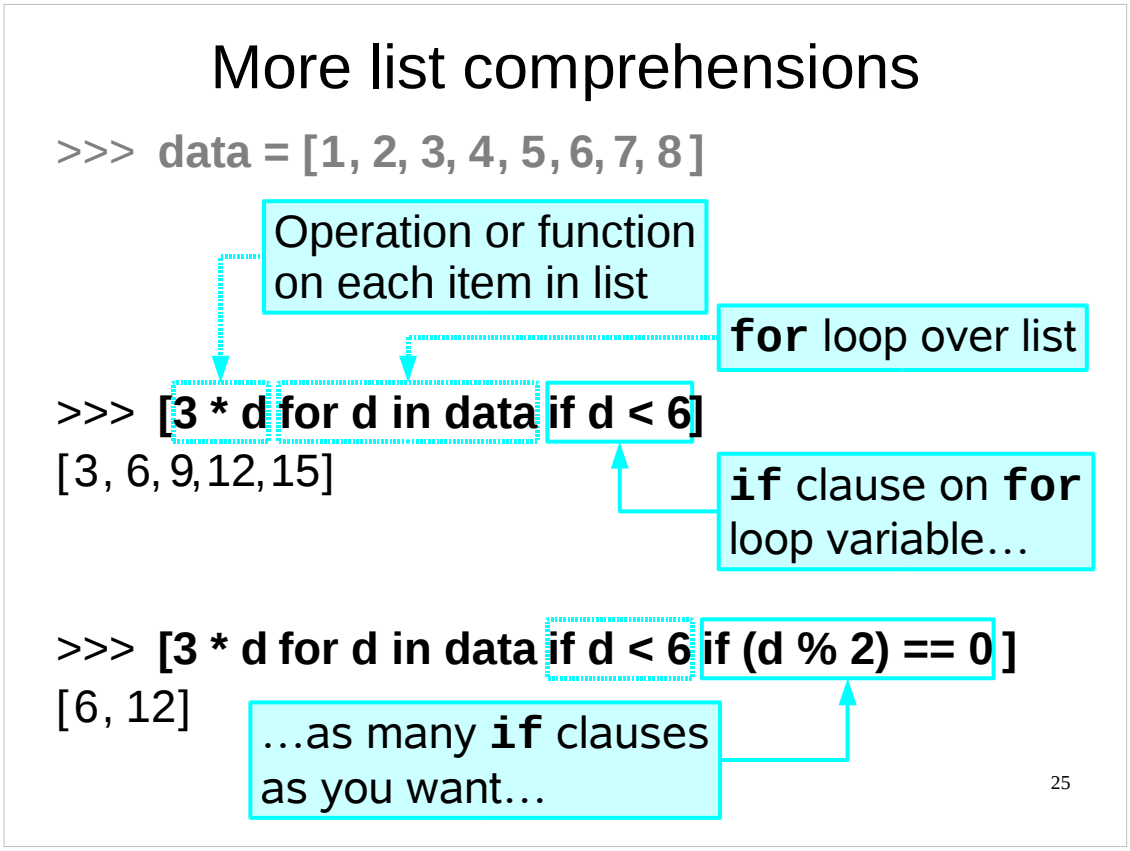
To repeat a list of *lists* (or other complicated data types), **don't** use list repetition, use a list comprehension instead.

24

So now we can sensibly repeat a list of lists (or other complicated data types).

As we see above, the "operation" that we carry out in our list comprehension can in fact be a constant value (which can be of any type: integers, floats, even lists (as above)) – this will create a new list, each of whose items is a ***copy*** (a "deep copy") of the specified value.

(Recall that the `range()` function gives me a list of integers from the first integer to one less than the second integer, so `range(0,2)` = `[0, 1]`.)



List comprehensions are even more versatile than you might at first imagine – as well as looping over an existing list, we can also add one or more `if` clauses to our list comprehension to further limit the items from the original list upon which we want our operation or function to act.

So the list comprehension

```
[3 * d for d in data if d < 6]
```

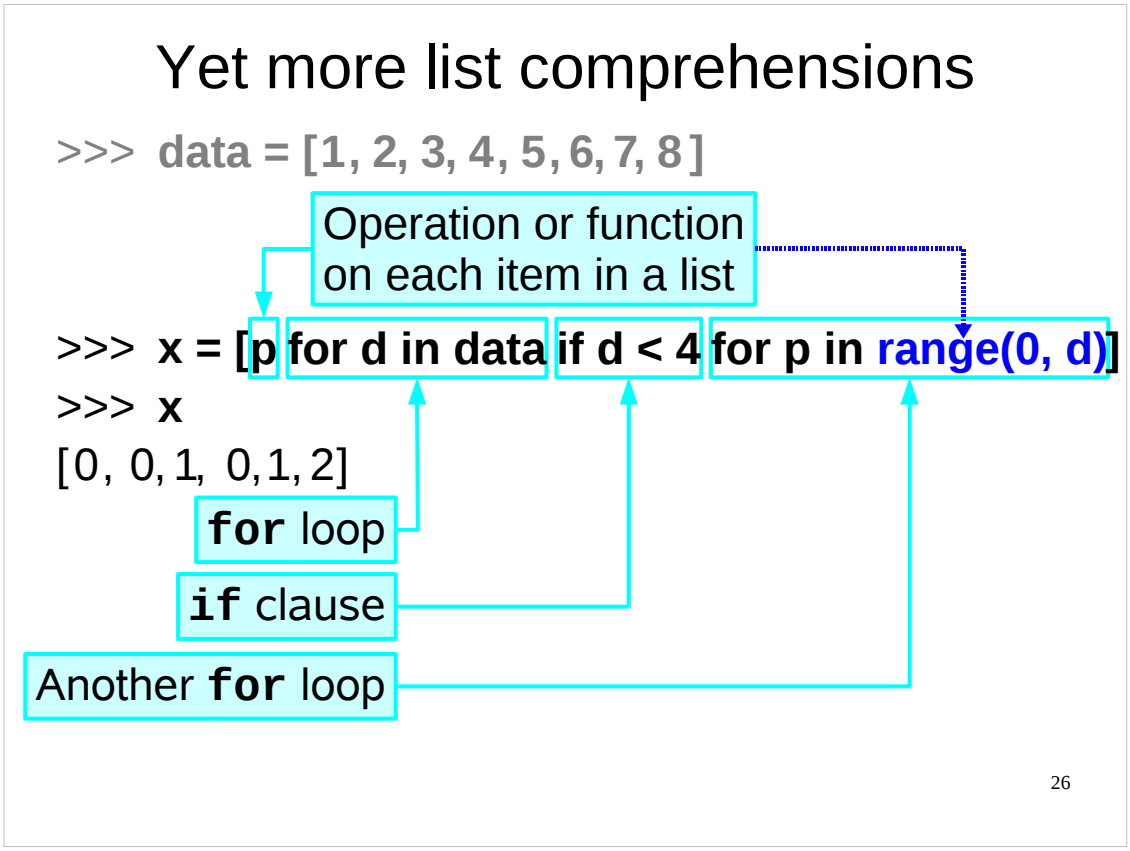
should be read as something like "`for` each item in the list `data` whose value is less than 6, multiply 3 by that item and add it to our new list".

And the list comprehension

```
[3 * d for d in data if d < 6 if (d % 2) == 0]
```

should be read as something like "`for` each item in the list `data` whose value is less than 6, `if` that item is divisible by 2, multiply 3 by that item and add it to our new list".

Recall that for integers `%` means "the (non-negative) remainder when divided by" (usually read as "mod", short for "modulo"), so the expression "`d % 2`" is only equal to 0 if `d` is even. (We don't actually need the brackets around the "`d % 2`" in the `if` clause, I've just put them in there for clarity.)



In fact, they are quite impressively versatile – as well as adding one or more `if` clauses to our list comprehension, we can also add one or more *additional* `for` loops. So, the general form of a list comprehension is:

`[` *"function or operation" "`for` loop" "zero or more `if` clauses and/or `for` loops"* `]`

So the list comprehension

```
[p for d in data if d < 4 for p in range(0,d)]
```

should be read as something like "`for` each *item* in the list `data` whose value is less than 4, loop over the temporary list `range(0,` *item*`)`, adding each item from this temporary list to our new list".

I.e. the following line of Python:

```
x = [p for d in data if d < 4 for p in range(0,d)]
```

is equivalent to:

```
x = []
for d in data:
    if d < 4:
        for p in range(0,d):
            x.append(p)
```

(Recall that the `range()` function gives me a list of integers from the first integer to one less than the second integer, so, for example, `range(0,3) = [0, 1, 2]`.)

Evaluate the following Python statements in your head. `primes` is defined as:

```
primes = [2, 3, 5, 7, 11, 13, 17, 19]
```

```
>>> [8 / p for p in primes]
>>> [p for p in primes if p % 3 > 0]
>>> [[0,0,0] for x in range(2,5)]
>>> [93 % p for p in primes if 93 % p != 0]
>>> [5 ** p for p in primes if p % 4 == 0]
>>> [2 * x for p in primes[0::2] for x in range(p-1,p+2)]
```

Now try them interactively in Python and see if you were correct.

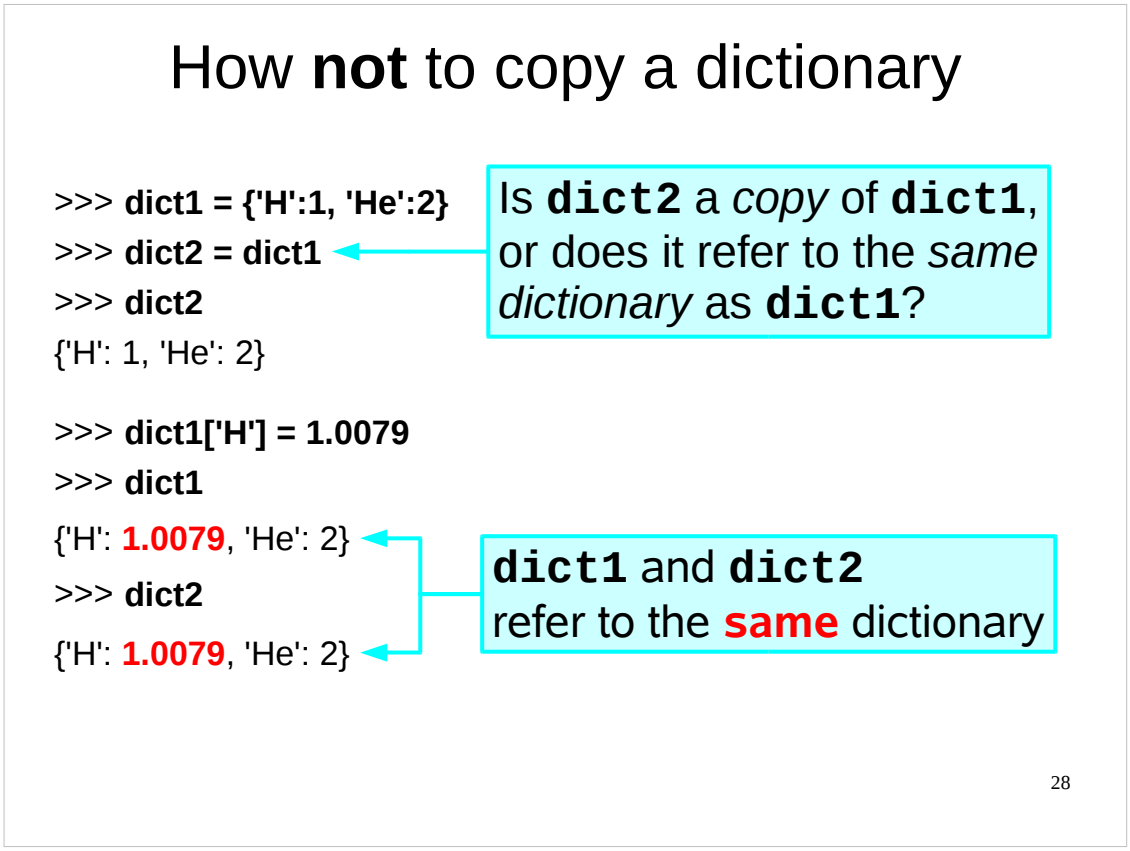
27

If you run into problems with this exercise, or if you don't understand any of the Python on the slide above, please ask the course giver or a demonstrator for help.

Recall that:

- For integers, `a % b` means "the (non-negative) remainder when `a` is divided by `b`" (usually read as "`a` mod `b`", "mod" being short for "modulo"); and
- The `range()` function gives me a list of integers from the first integer to one less than the second integer, so `range(6,9)` = `[6, 7, 8]`.

When you've finished take a short break of one or two minutes – remember that, in this context, "break" means "break from using the computer".



We've seen how we can "properly" copy a list. What about if we want to copy a dictionary?

If we've assigned a dictionary to a variable (say a variable called `dict1`) and we want to make a copy of that dictionary (and assign that copy to another variable, say a variable called `dict2`), we might be tempted to do something like this:

```
dict2 = dict1
```

Unfortunately, as with lists, this does ***not*** work in the way we might expect!

What happens is that both `dict1` and `dict2` now refer to the same dictionary in the computer's memory. Changing `dict1` will affect `dict2` (and vice-versa), since they are both actually the same dictionary. When we "copy" a dictionary like this, we don't actually copy it at all, we just create a new variable that "points" to the same dictionary that we had before. (This is sometimes called a "shallow copy".)

Again, we can see that this is the case if we use the `id()` function. If you've typed in the Python on the slide above, you can try the `id()` function on `dict1` and see what it returns:

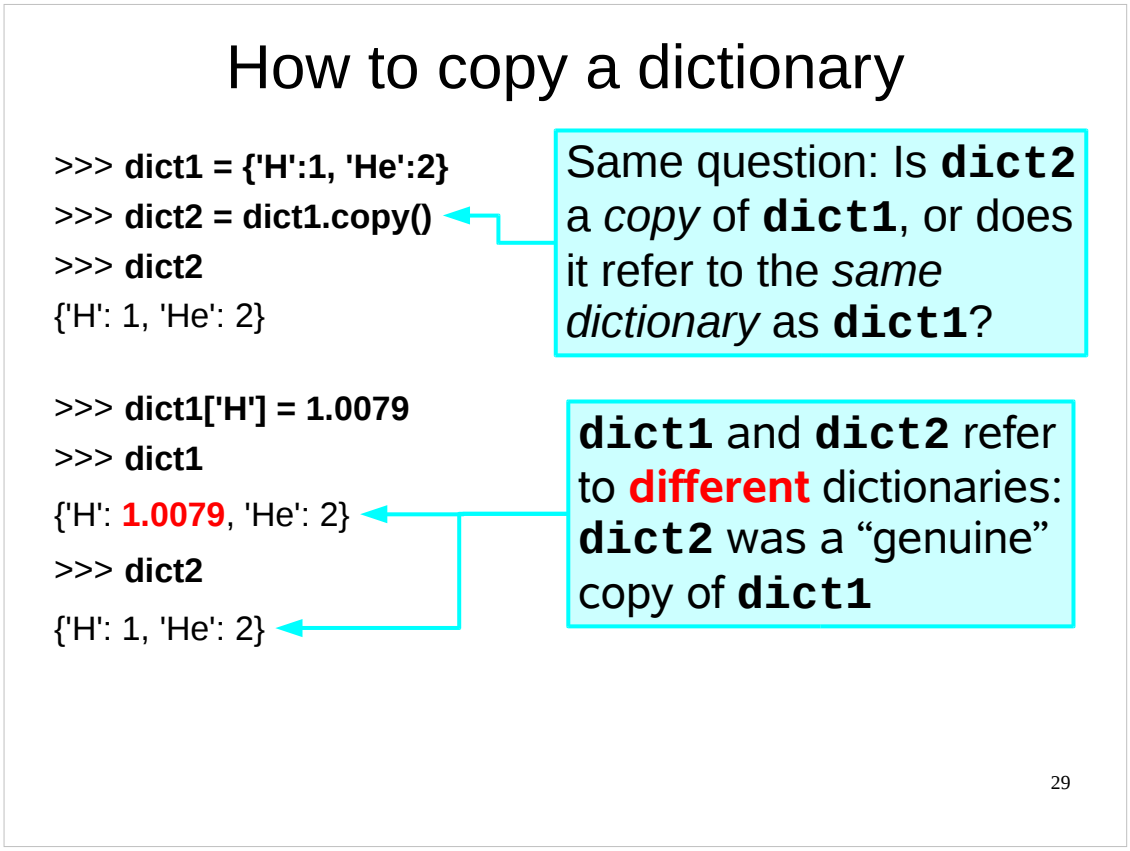
```
>>> id(dict1)
```

and then on `dict2`:

```
>>> id(dict2)
```

You should find that `id()` returns the same value for both these variables (whatever that value might happen to be).

So how *can* we make a real copy of a dictionary?…



…Well, fortunately, dictionaries provide a method, the `copy()` method, that allows us to do just that: create a real, genuine, honest-to-goodness **copy** that is a *different* dictionary (with the same key/value pairs), stored in a *different* memory location. (This is sometimes called a "deep copy".) As `copy()` is a method of dictionaries, we can use it on any dictionary – it returns a *copy* of the dictionary:

```
>>> {'H':1, 'He':2}.copy()
{'H': 1, 'He': 2}
```

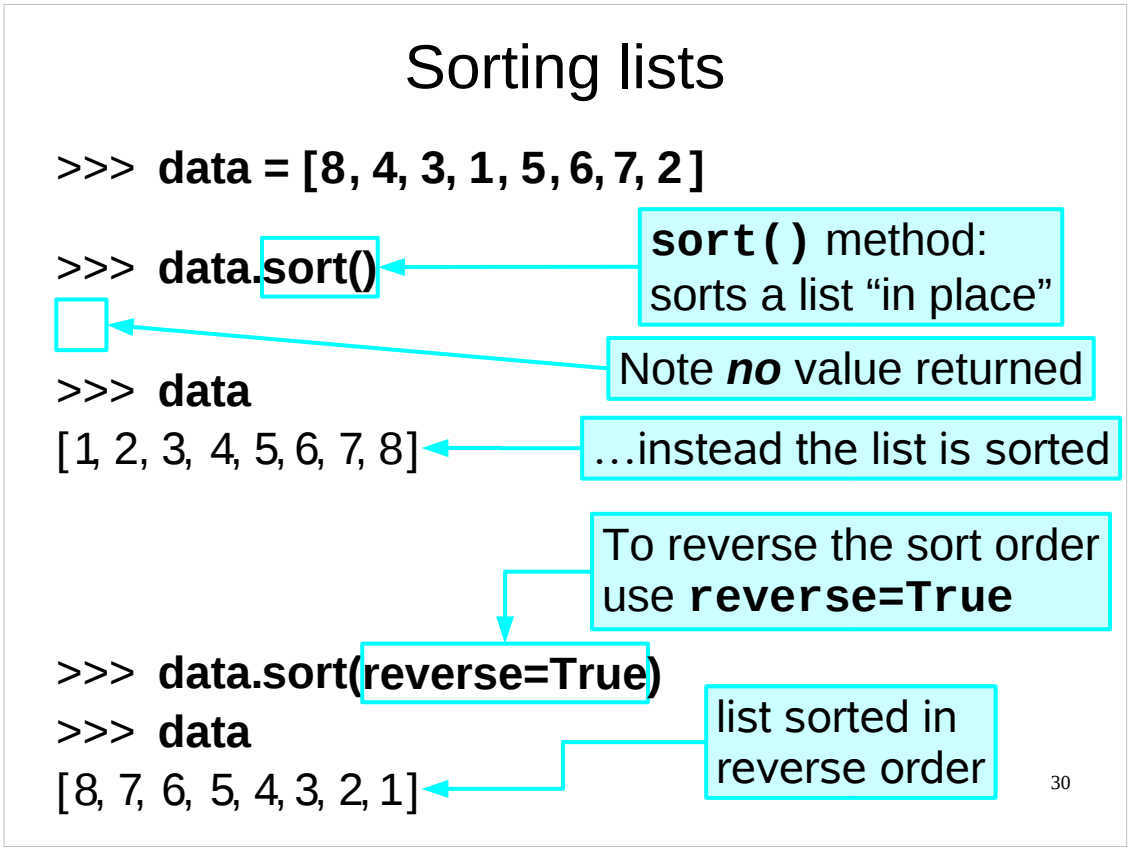
Again, we can see that this is the case using the `id()` function. If you've typed in the Python on the slide above, you can try the `id()` function on `dict1` and see what it returns:

```
>>> id(dict1)
```

and then on `dict2`:

```
>>> id(dict2)
```

You should find that `id()` returns *different* values for each variable (whatever those values might happen to be). That means that they refer to different objects in memory (which may or may not happen to have the same value).



Another method that lists possess is the `sort()` method. This sorts a list “in place”.

This method also provides a quick way to reverse the sort order: call the `sort()` method setting the `reverse` named argument to the Boolean `True` (i.e. call the method using `sort(reverse=True)` rather than just `sort()`). Note that the `reverse` named argument was introduced in Python 2.4, so you can’t use it in earlier versions of Python.

Note that lists also have a `reverse()` method that does ***not*** do a reverse sort of the list, but rather reverses (“in place”) the order of the items in the list:

```
>>> data = [8, 4, 3, 1, 5, 6, 7, 2]
>>> data.reverse()
>>> data
[2, 7, 6, 5, 1, 3, 4, 8]
```

(Obviously, this means you could also do a reverse sort of a list by calling the `sort()` method immediately followed by the `reverse()` method, but it is easier and much more efficient to just call the `sort()` method with `reverse=True`.)

The `sort()` method also allows you to define your own sort order for sorting a list – you do this by using defining a comparison function and giving that function to the `sort()` method as an argument. For further details see the Python Library Reference sub-section on “Mutable Sequence Types”:

http://docs.python.org/library/stdtypes.html#typesseq-mutable

Exercise

Write a function that takes a dictionary and **`print`**s out its values in ascending order.
Dictionary → values in ascending order

{'Ar': 39.95,
 'H': 1.0079, →
 'N': 14.007}

```
1.0079
14.007
39.95
```

…since if we arrange the values of the above dictionary in ascending order, they look like this: 1.0079, 14.007, 39.95

31

So if the function took as its input the dictionary:

```
{'Ar': 39.95, 'H': 1.0079, 'N': 14.007}
```

it would produce the output:

```
1.0079
14.007
39.95
```

If you run into problems with this exercise, ask the course giver or a demonstrator for help.

(An answer is given on the page after next.)

Hint: Recall that if `x` is a dictionary then `x.keys()` gives you a list of the dictionary's keys (in a might as well be random order) whilst `x.values()` gives you a list of the values in the dictionary (also in a (possibly different) might as well be random order).

Exercise *redux*

Write a function that takes a dictionary and **`print`**s out its values in *descending* order of the corresponding **keys**.
Dictionary → values in descending order of *keys*

{'H': 1.0079, 'N': 14.007, 'Ar': 39.95} → `14.007` `1.0079` `39.95`

…since if we arrange the keys of the above dictionary in descending order, they look like this:
'N', 'H', 'Ar'

32

So if the function took as its input the dictionary:

```
{'H': 1.0079, 'N': 14.007, 'Ar': 39.95}
```

it would produce the output:

```
14.007
1.0079
39.95
```

If you run into problems with this exercise, ask the course giver or a demonstrator for help.

After this exercise take at least a 5 or 10 minute break. Remember that this means you should **stop** using the computer, and move around, exercise your arms, wrists, neck, etc.

(An answer is given to this exercise on the page after next.)

Hint: Recall that if `x` is a dictionary then `x.keys()` gives you a list of the dictionary's keys (in a might as well be random order) whilst `x.values()` gives you a list of the values in the dictionary (also in a (possibly different) might as well be random order).

Answer to Exercise

```
def print_dict_values_sorted(dict):

    sorted_values = dict.values()
    sorted_values.sort()

    for value in sorted_values:
        print value
```

33

Here is a solution to the first exercise that you were to attempt over the break.

If there is anything in the solution that you do not understand, or if your solution looks utterly different from that shown above, please tell the course giver or demonstrator.

Answer to Exercise *redux*

```
def print_dict_values_sorted_by_reverse_keys(dict):

    ordered_keys = dict.keys()
    ordered_keys.sort(reverse=True)

    for key in ordered_keys:
            print dict[key]
```


Here is a solution to the second exercise that you were to attempt over the break.

If there is anything in the solution that you do not understand, or if your solution looks utterly different from that shown above, please tell the course giver or demonstrator.

Temporary files

Temporary files: *a great way to accidentally give access to your system to someone who shouldn't have it.*

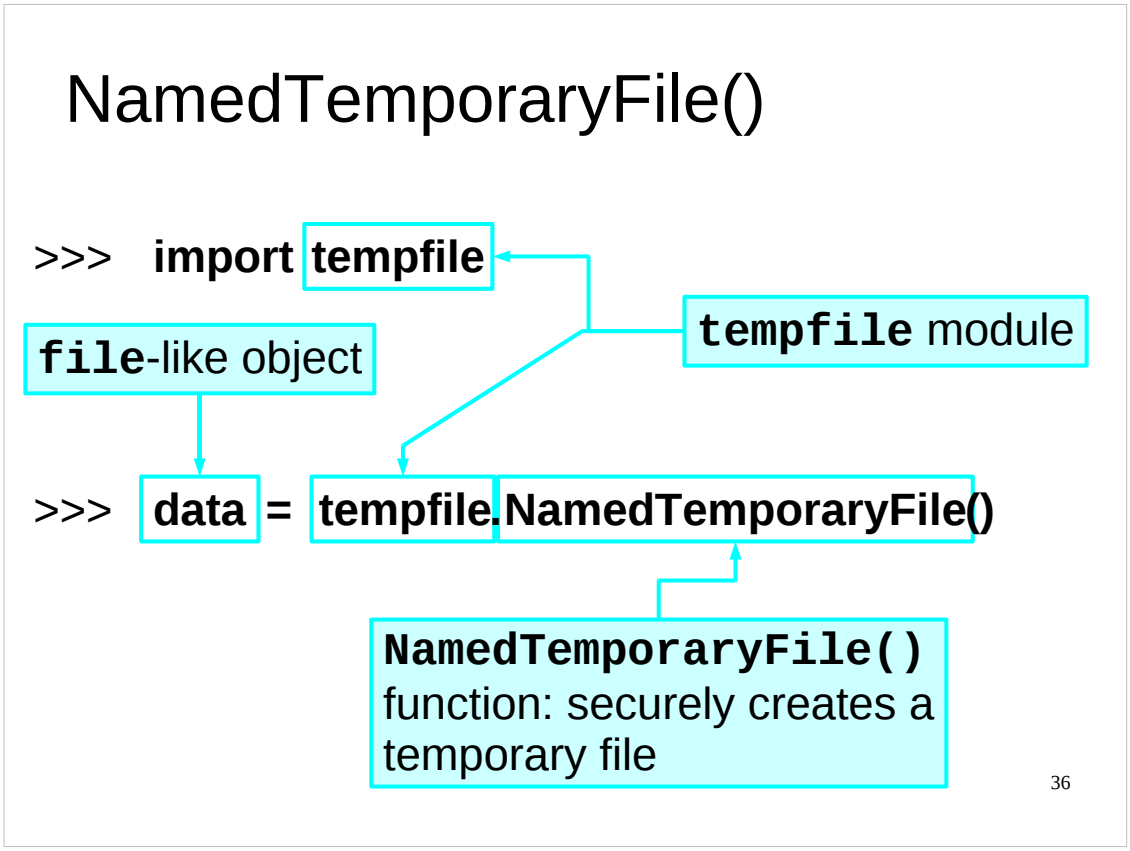
35

We're going to briefly return now to file I/O to look at one particular aspect of it: temporary files.

Often we need a file to write some data to for a short period of time, which we will then delete. We might need to do this because we need to pass some intermediate data to another program for processing but we don't want to keep that intermediate data.

Some of you may think: "but I already know how to create a file, why don't I just create a temporary file myself?". In general, that's an extremely ***bad*** idea – on a multi-user system it is very difficult to securely create a temporary file, and very easy to insecurely create one, which, over the years, has led to any number of security holes in systems that have allowed unauthorised people to get access to the system.

Fortunately, there are a number of functions that have been provided which do this for us in a safe, secure manner. We'll look at two of them now.



The `NamedTemporaryFile()` function (which lives in the `tempfile` module) will securely create a temporary file for us, which it will delete when we close the file. It returns a `file`-like object ("like" as in it has all the familiar properties and methods of `file` objects, but it is actually a different type of object). This function was introduced in Python 2.3, so you can't use it in earlier versions of Python.

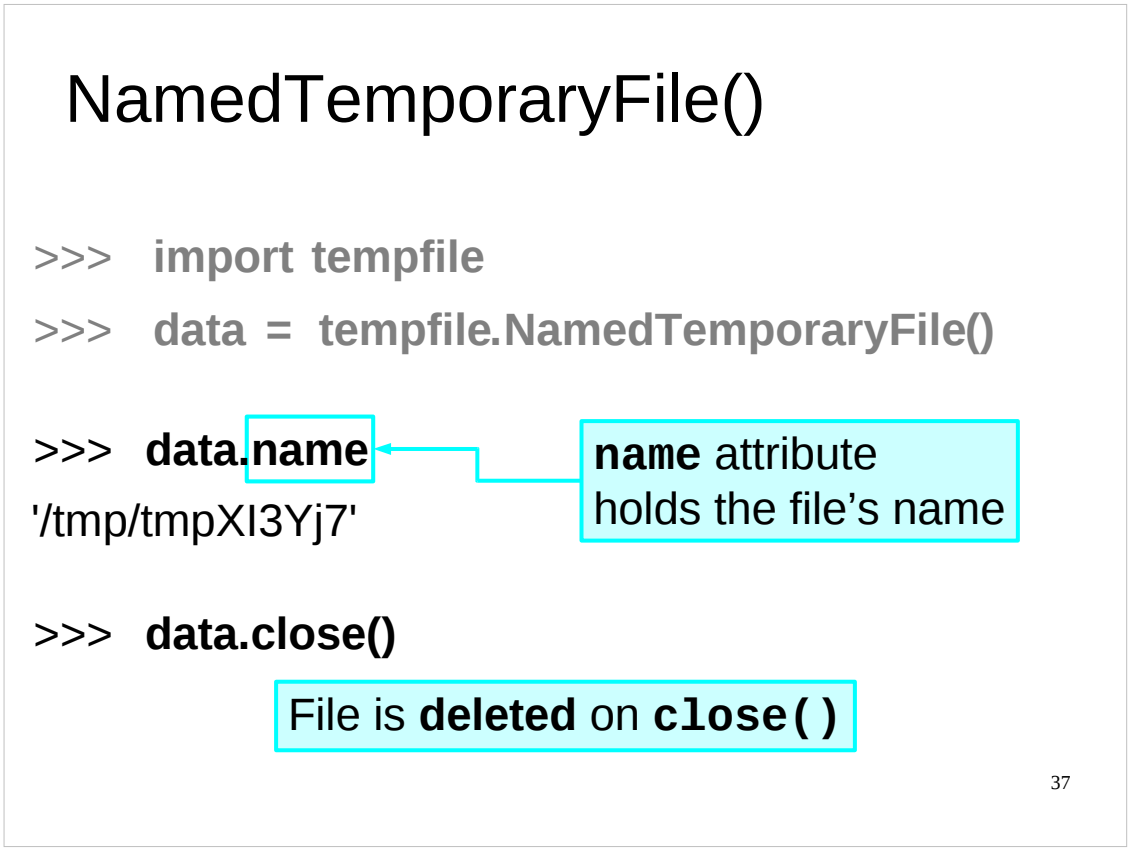
The temporary file is opened in binary mode, and also is opened for both reading and writing (this is a a new mode we haven't yet met, which is specified by using `'w+b'` – `'w+'` specifies the file should be opened for *both* reading and writing, the `'b'` on the end specifies it should be opened in **b**inary mode). Note that if the file already exists, opening it in `'w+'` mode will remove its contents (just as opening it in ordinary `'w'` mode does). Since this is a temporary file specially created for us this doesn't matter.

If, however, you want the temporary file opened in a different mode, then you can specify a mode to the `NamedTemporaryFile()` function, like this:

```
tempfile.NamedTemporaryFile(mode='w')
```

which would create a temporary file for **w**riting (in text mode).

Note that `NamedTemporaryFile()` will *delete* the temporary file when we close it.



The name of the temporary file, in case this is of interest, lives in the `name` attribute of the `file`-like object created by the `NamedTemporaryFile()` function.

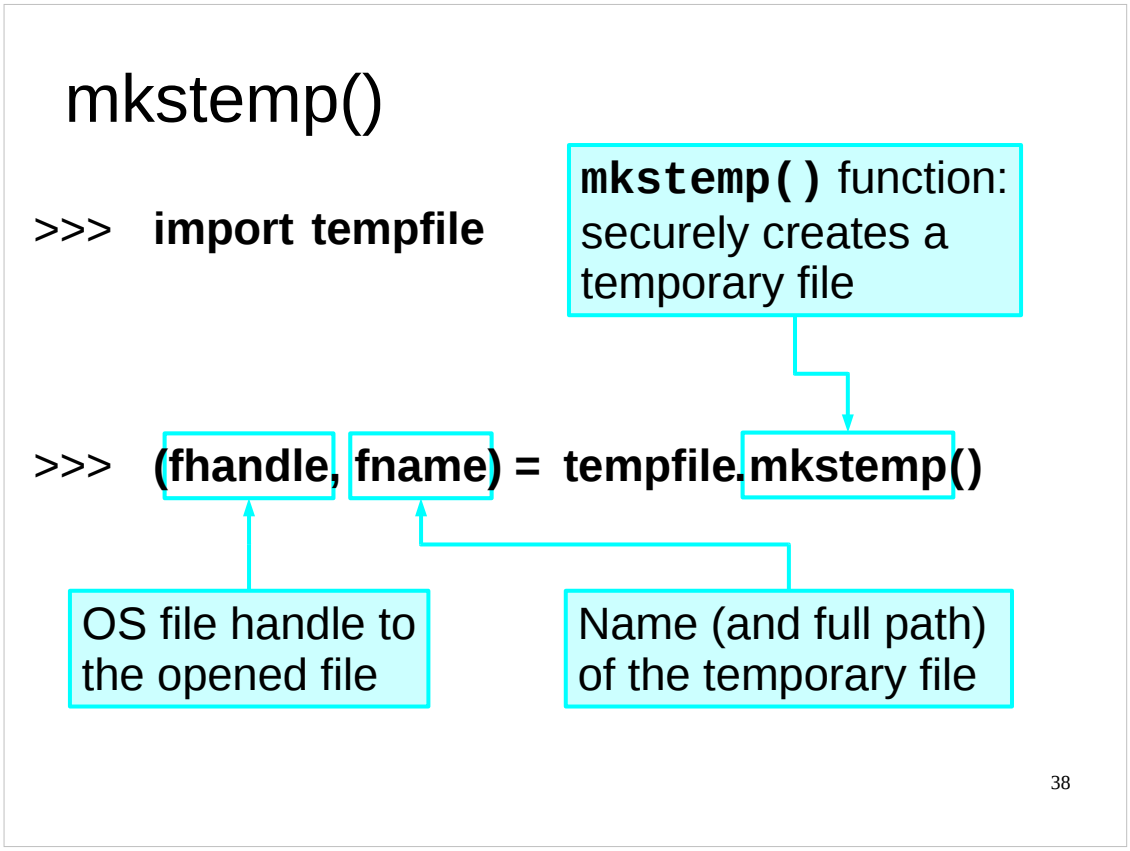
Now, `NamedTemporaryFile()` will delete the temporary file when we close it, which might not be what we want if, for instance, we want to create a temporary file to pass to another program. So how can we securely create a temporary file without having it automatically deleted?…

(Note that if you try the Python commands above, you will almost certainly get a completely different file name for the temporary file.)

(Finally, note that, starting with Python 2.6, `NamedTemporaryFile()` has a named argument, `delete`, that you can set to `False` when calling `NamedTemporaryFile()` if you do not want the temporary file to be deleted when it is closed, like this:

```
tempfile.NamedTemporaryFile(delete=False)
```

Unfortunately, this functionality does not exist in versions of Python prior to Python 2.6, so you can't do this if you are using a version of Python earlier than 2.6.)

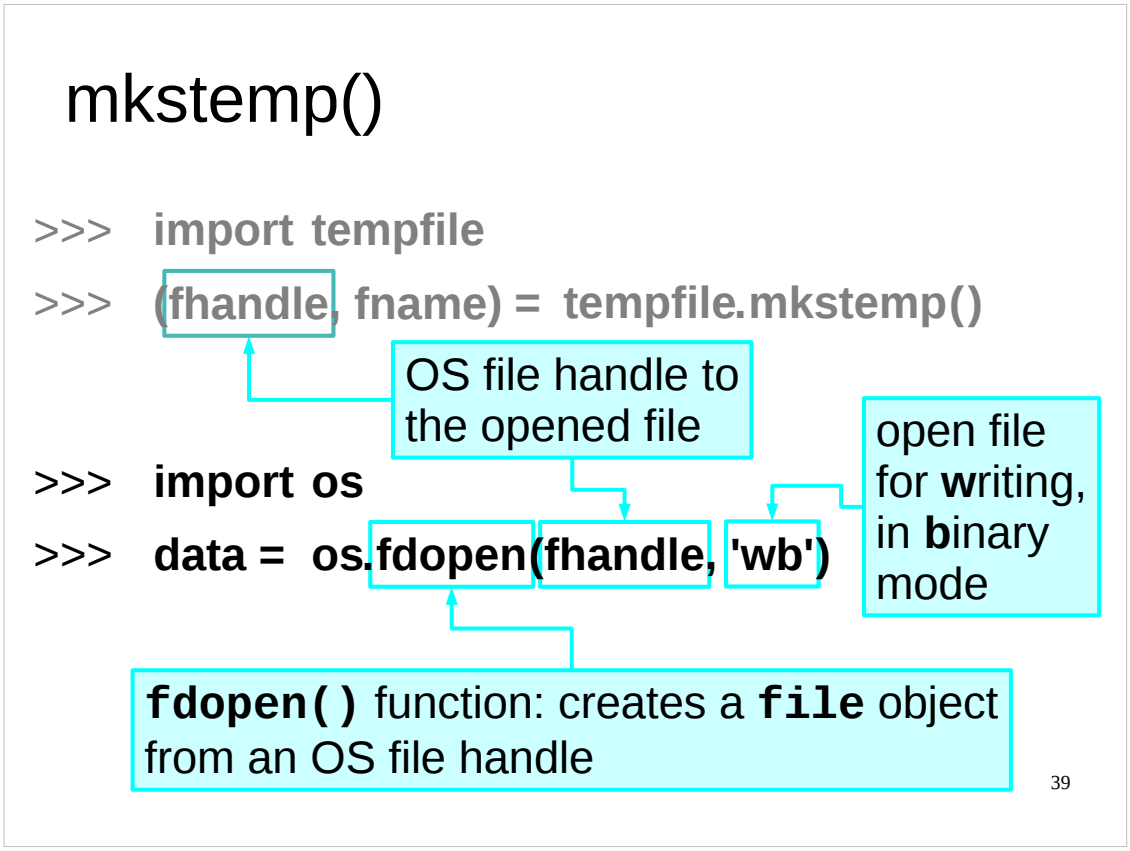


The `mkstemp()` function (which also lives in the `tempfile` module) will securely create and open for both reading and writing (in binary mode) a temporary file for us, but having created it, it leaves it alone. It is up to us to delete it when we've finished using it. The `mkstemp()` function returns a tuple consisting of a *file handle* to the opened file, and the file's name (and full path), as a string. If you want `mkstemp()` to open the file in text mode, set the named argument `text` to `True` when calling `mkstemp()`, like this:

```
tempfile.mkstemp(text=True)
```

(Note that the `mkstemp()` function was also introduced in Python 2.3, so you can't use it in earlier versions of Python.)

The file handle is ***not*** a `file` object, and so does not have all the useful `file` object methods. Instead it provides low level operating system (OS) access to the file, which is not something we wish to use if we can help it. Consequently the best thing to do with this file handle is use it to create a Python `file` object, after which we can forget about it and just use the familiar Python `file` object methods. How do we do that…?



To create a Python `file` object from a file handle we need to use the `fdopen()` function that lives in the `os` module. If we give the `os.fdopen()` function an open file handle, it will create a corresponding Python `file` object for us, created with the specified mode (if we don't specify a mode it behaves as though we specified a mode of `'r'`). (The mode that we give to `os.fdopen()` is the same as we would give to the `open()` command, except that it *must* start with an `'r'`, `'w'` or `'a'`).

The mode we give `os.fdopen()` **must** be compatible with the mode which was used when creating the file handle. So, if `tempfile.mkstemp()` has opened the file in **b**inary mode (its default behaviour), then we should tell `os.fdopen()` to do likewise (i.e. add a `'b'` to the end of the mode we give `os.fdopen()`). Similarly, if `tempfile.mkstemp()` has opened the file in text mode, we should tell `os.fdopen()` to do likewise (no `'b'`).

Once we've created a `file` object for our newly created temporary file, we can get on with accessing it in the normal Python manner (using the `write()` method, etc). ***Remember*** to `close` the file using the `file` object's `close()` method when you've finished using it!

It is only when a file is closed that the writes to it are committed to the file system.

Saving complex objects to a file

Object serializiation: **`pickle`** and **`cPickle`** modules

40

Python has two modules which can be used for what is sometimes called "object serialization", which is also known – in the Python world – as "pickling". This is essentially a way of taking a Python object and storing it in a compact format (usually on disk). ("Serialization" is also known as "marshalling" or "flattening", although Python uses the term "marshalling" in a more specialised manner.)

Python can pickle almost all its basic object types – integers, long integers, floating point numbers, complex numbers, Booleans, the `NoneType`, strings, etc – and, more usefully, many of its composite data types – such as lists, tuples and dictionaries – provided all their individual items are also objects it can pickle. Thus, if, for example, your dictionary contains only integers, floating point numbers, etc, or lists or tuples of such objects, then you can pickle it. This provides a very easy way of storing a complex object like a dictionary or a list of lists without you having to individually write each item the object contains out to a file. You can find the complete list of objects that can be pickled in the "What can be pickled and unpickled?" subsection of the `pickle` module's documentation:

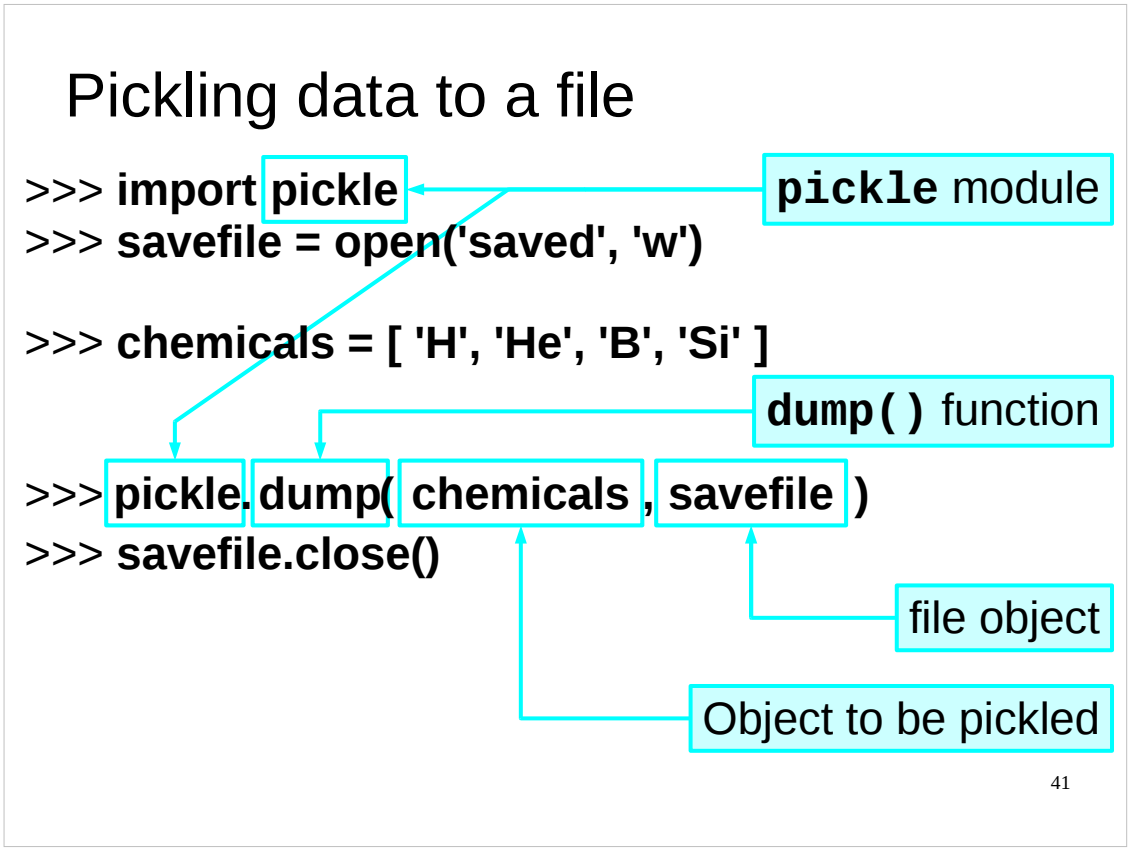
http://docs.python.org/library/pickle.html

There are two modules which you can use almost interchangeably for pickling – the `pickle` module and the `cPickle` module. Why are there two of them? Well, the `cPickle` module is implemented in C and so is much, much faster than the `pickle` module. However, the `pickle` module can be extended using Python's object oriented framework (not covered in this course). So if you have some special requirement that can't be satisfied by the built-in `pickle` module, you might want to extend it – which you can't do with the `cPickle` module. Most users don't need to do this though, and so can use the `cPickle` module (and gain the benefit of its speed).

Python guarantees that if you use the `pickle` module to store something, you can load it again using the `cPickle` module, and vice-versa. In addition, if you pickle something on one machine, you can load it again on a different machine, even if that machine is running a different operating system or has a different version of Python (well, provided the versions of Python aren't *too* different).

The `pickle` and `cPickle` modules are covered in more detail in the "Python: Checkpointing" course:

http://www.training.cam.ac.uk/ucs/course/ucs-pythonchckpt



As previously mentioned, Python can pickle almost all its basic object types – integers, floating point numbers, strings, etc, and, more usefully, many of its composite data types – such as lists, tuples and dictionaries – provided all their individual items are also objects it can pickle. Thus, if, for example, your dictionary contains only integers, floating point numbers, etc, or lists or tuples of such objects, then you can pickle it.

The basic way of "pickling" data to a file is to use the `dump()` function. The `dump()` function works on `file` objects, so you need to `open` the file (for writing) before calling the `dump()` function.

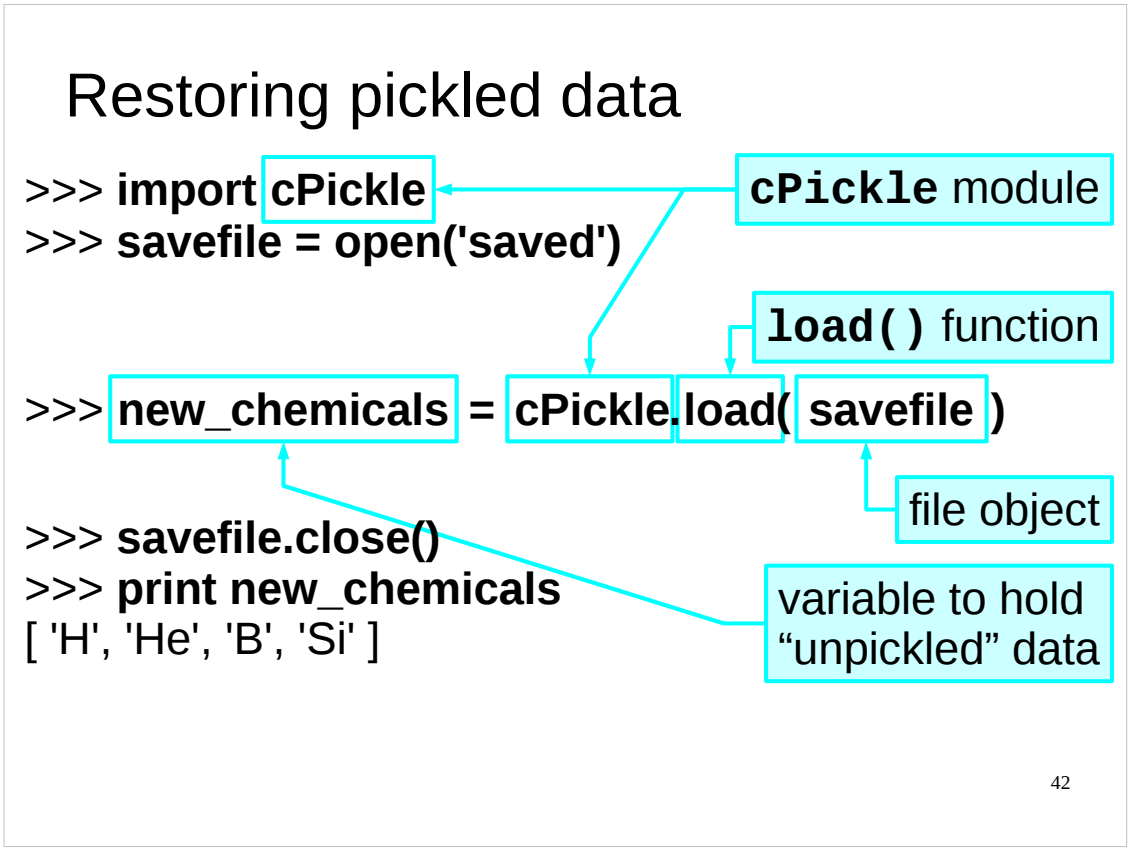
As mentioned before, you can use the `dump()` function from either the `pickle` or the `cPickle` module.

If you give the `dump()` function something that it cannot pickle, Python will raise a `PicklingError` exception (as this exception is defined in the `pickle` and `cPickle` modules, if you wish to handle it you would refer to it as `pickle.PicklingError` or `cPickle.PicklingError`). In rare cases, attempting to pickle a very complex data structure may cause a `RuntimeError` exception to be raised.

Remember to `close` the file to which you are pickling using its `close()` method when you've finished using it.

It is only when a file is closed that the writes to it are committed to the file system.

You normally only store a single "pickle" of data in a file. If you need to pickle several pieces of data and store them in the same file, just put all the data into a tuple and pickle the tuple. The author knows of no good reason to store multiple "pickles" of data in a single file. If, however, you are absolutely convinced you need to do this, then have a look at the `shelve` module (one of the standard Python modules).



The basic way of restoring pickled data from a file is to use the `load()` function. The `load()` function works on `file` objects, so you need to `open` the file (for reading) before calling the `load()` function. (Obviously, once you've restored the pickled data, make sure you `close` the file.)

As mentioned before, you can use the `load()` function from either the `pickle` or the `cPickle` module.

If the `load()` function has a problem with unpickling the data, Python will usually raise an `UnpicklingError` exception. (Note that as this exception is defined in the `pickle` and `cPickle` modules, if you wish to handle it you would refer to it as `pickle.UnpicklingError` or `cPickle.UnpicklingError`.) However, there are a number of other exceptions that might be raised instead if there is a problem unpickling the data depending on exactly what the problem was. Some of the other exceptions that Python might raise when there is a problem unpickling data include (but are not limited to) the `AttributeError`, `EOFError`, `ImportError` or `IndexError` exceptions.

Structuring a program for checkpointing

1. Initialise
2. Check for the existence of a previous checkpoint:
    1. If present, load it
3. Start processing loop:
    1. If there's a checkpoint file, retrieve state from file
    2. Process data
    3. Save state to checkpoint file
4. Final output

43

The most common use of pickling is for *checkpointing*.

Basically, you put all the variables that hold the current state of your program (i.e. all the variables whose values you would need if you wanted to restart the program whatever point it has just reached) into a tuple and then pickle that tuple. The file that contains this pickled data is your *checkpoint file*.

Each time you have done a certain amount of processing you dump the state of your program out to a checkpoint file. Then you restore from the checkpoint, i.e. load the pickled data from the checkpoint file – so you can be sure that the checkpoint actually did correctly store all the data it should have – and continue.

We examine this process in more detail in the "Python: Checkpointing" course:

http://www.training.cam.ac.uk/ucs/course/ucs-pythonchckpt