50 ESSENTIAL PYTHON COMMANDS

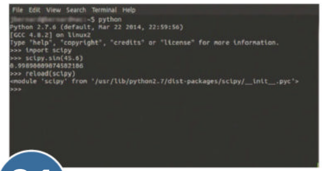
Python is known as a very dense language, with lots of modules capable of doing almost anything. Here, we will look at the core essentials that everyone needs to know

Python has a massive environment of extra modules that can provide functionality in hundreds of different disciplines. However, every programming language has a core set of functionality that everyone should know in order to get useful work done. Python is no different in this regard. Here, we will look at 50 commands that we consider to be essential to programming in Python. Others may pick a slightly different set, but this list contains the best of the best.

We will cover all of the basic commands, from importing extra modules at the beginning of a program to returning values to the calling environment at the end. We will also be looking at some commands that are useful in learning about the current session within Python, like the current list of variables that have been defined and how memory is being used.

Because the Python environment involves using a lot of extra modules, we will also look at a few commands that are strictly outside of Python. We will see how to install external modules and how to manage multiple environments for different development projects. Since this is going to be a list of commands, there is the assumption that you already know the basics of how to use loops and conditional structures. This piece is designed to help you remember commands that you know you've seen before, and hopefully introduce you to a few that you may not have seen yet.

Although we've done our best to pack everything you could ever need into 50 tips, Python is such an expansive language that some commands will have been left out. Make some time to learn about the ones that we didn't cover here, once you've mastered these.



```
[redacted]~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import scipy
>>> scipy.sin(45.6)
0.99890009874582186
>>> reload(scipy)
<module 'scipy' from '/usr/lib/python2.7/dist-packages/scipy/__init__.pyc'>
>>>
```

01 Importing modules

The strength of Python is its ability to be extended through modules. The first step in many programs is to import those modules that you need. The simplest import statement is to just call 'import modulename'. In this case, those functions and objects provided are not in the general namespace. You need to call them using the complete name (modulename.methodname). You can shorten the 'modulename' part with the command 'import modulename as mn'. You can skip this issue completely with the command 'from modulename import *' to import everything from the given module. Then you can call those provided capabilities directly. If you only need a few of the provided items, you can import them selectively by replacing the '*' with the method or object names.

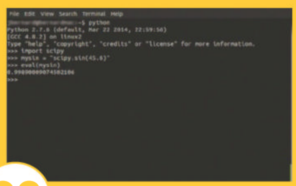
02 Reloading modules

When a module is first imported, any initialisation functions are run at that time. This may involve creating data objects, or initiating connections. But, this is only done the first time within a given session. Importing the same module again won't re-execute any of the initialisation code. If you want to have this code re-run, you need to use the reload command. The format is 'reload(modulename)'. Something to keep in mind is that the dictionary from the previous import isn't dumped, but only written over. This means that any definitions that have changed between the import and the reload are updated correctly. But if you delete a definition, the old one will stick around and still be accessible. There may be other side effects, so always use with caution.

03 Installing new modules

While most of the commands we are looking at are Python commands that are to be executed within a Python session, there are a few essential commands that need to be executed outside of Python. The first of these is pip. Installing a module involves downloading the source code, and compiling any included external code. Luckily, there is a repository of hundreds of Python modules available at **http://pypi.python.org**. Instead of doing everything manually, you can install a new module by using the command 'pip install modulename'. This command will also do a dependency check and install any missing modules before installing the one you requested. You may need administrator rights if you want this new module installed in the global library for your computer. On a Linux machine, you would simply run the pip command with sudo. Otherwise, you can install it to your personal library directory by adding the command line option '—user'.

"Every programming language out there has a core set of functionality that everyone should know in order to get useful work done. Python is no different"



```
[redacted]~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import scipy
>>> mysin = "scipy.sin(45.6)"
>>> eval(mysin)
0.99890009874582186
>>>
```

04 Executing a script

Importing a module does run the code within the module file, but does it through the module maintenance code within the Python engine. This maintenance code also deals with running initialising code. If you only wish to take a Python script and execute the raw code within the current session, you can use the 'execfile("filename.py")' command, where the main option is a string containing the Python file to load and execute. By default, any definitions are loaded into the locals and globals of the current session. You can optionally include two extra parameters the execfile command. These two options are both dictionaries, one for a different set of locals and a different set of globals. If you only hand in one dictionary, it is assumed to be a globals dictionary. The return value of this command is None.

05 An enhanced shell

The default interactive shell is provided through the command 'python', but is rather limited. An enhanced shell is provided by the command 'ipython'. It provides a lot of extra functionality to the code developer. A thorough history system is available, giving you access to not only commands from the current session, but also from previous sessions. There are also magic commands that provide enhanced ways of interacting with the current Python session. For more complex interactions, you can create and use macros. You can also easily peek into the memory of the Python session and decompile Python code. You can even create profiles that allow you to handle initialisation steps that you may need to do every time you use iPython.

06 Evaluating code

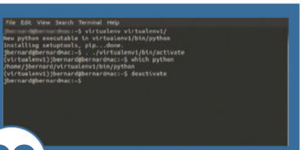
Sometimes, you may have chunks of code that are put together programmatically. If these pieces of code are put together as a string, you can execute the result with the command 'eval("code_string")'. Any syntax errors within the code string are reported as exceptions. By default, this code is executed within the current session, using the current globals and locals dictionaries. The 'eval' command can also take two other optional parameters, where you can provide a different set of dictionaries for the globals and locals. If there is only one additional parameter, then it is assumed to be a globals dictionary. You can optionally hand in a code object that is created with the compile command instead of the code string. The return value of this command is None.

07 Asserting values

At some point, we all need to debug some piece of code we are trying to write. One of the tools useful in this is the concept of an assertion. The assert command takes a Python expression and checks to see if it is true. If so, then execution continues as normal. If it is not true, then an AssertionError is raised. This way, you can check to make sure that invariants within your code stay invariant. By doing so, you can check assumptions made within your code. You can optionally include a second parameter to the assert command. This second parameter is Python expression that is executed if the assertion fails. Usually, this is some type of detailed error message that gets printed out. Or, you may want to include cleanup code that tries to recover from the failed assertion.

08 Mapping functions

A common task that is done in modern programs is to map a given computation to an entire list of elements. Python provides the command 'map()' to do just this. Map returns a list of the results of the function applied to each element of an iterable object. Map can actually take more than one function and more than one iterable object. If it is given more than one function, then a list of tuples is returned, with each element of the tuple containing the results from each function. If there is more than one iterable handed in, then map assumes that the functions take more than one input parameter, so it will take them from the given iterables. This has the implicit assumption that the iterables are all of the same size, and that they are all necessary as parameters for the given function.

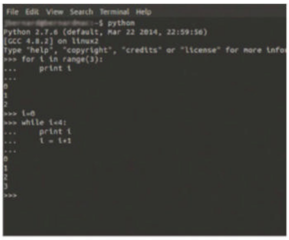


```
File  Edit  View  Search  Terminal  Help
[redacted]:~$ virtualenv virtualenv1/
New python executable in virtualenv1/bin/python
Installing setuptools, pip...done.
jbernard@bernardmac:~$ . ./virtualenv1/bin/activate
(virtualenv1)jbernard@bernardmac:~$ which python
/home/jbernard/virtualenv1/bin/python
(virtualenv1)jbernard@bernardmac:~$ deactivate
jbernard@bernardmac:~$
```

09 Virtualenvs

Because of the potential complexity of the Python environment, it is sometimes best to set up a clean environment within which to install only the modules you need for a given project. In this case, you can use the virtualenv command to initialise such an environment. If you create a directory named 'ENV', you can create a new environment with the command 'virtualenv ENV'. This will create the subdirectories bin, lib and include, and populate them with an initial environment. You can then start using this new environment by sourcing the script 'ENV/bin/activate', which will change several environment variables, such as the PATH. When you are done, you can source the script 'ENV/bin/deactivate' to reset your shell's environment back to its previous condition. In this way, you can have environments that only have the modules you need for a given set of tasks.

"While not strictly commands, everyone needs to know how to deal with loops. The two main types of loops are a fixed number of iterations loop (for) and a conditional loop (while)"



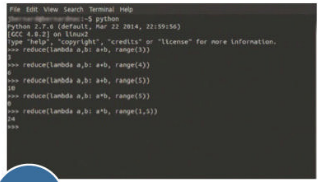
```
File  Edit  View  Search  Terminal  Help
[redacted]:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more info
>>> for i in range(3):
...     print i
...
0
1
2
>>> i=0
>>> while i<4:
...     print i
...     i = i+1
...
0
1
2
3
>>>
```

10 Loops

While not strictly commands, everyone needs to know how to deal with loops. The two main types of loops are a fixed number of iterations loop (for) and a conditional loop (while). In a for loop, you iterate over some sequence of values, pulling them off the list one at a time and putting them in a temporary variable. You continue until either you have processed every element or you have hit a break command. In a while loop, you continue going through the loop as long as some test expression evaluates to True. While loops can also be exited early by using the break command, you can also skip pieces of code within either loop by using a continue command to selectively stop this current iteration and move on to the next one.

11 Filtering

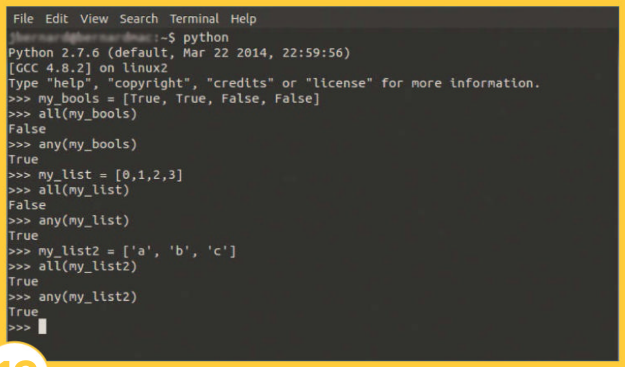
Where the command map returns a result for every element in an iterable, filter only returns a result if the function returns a True value. This means that you can create a new list of elements where only the elements that satisfy some condition are used. As an example, if your function checked that the values were numbers between 0 and 10, then it would create a new list with no negative numbers and no numbers above 10. This could be accomplished with a for loop, but this method is much cleaner. If the function provided to filter is 'None', then it is assumed to be the identity function. This means that only those elements that evaluate to True are returned as part of the new list. There are iterable versions of filter available in the itertools module.



```
File  Edit  View  Search  Terminal  Help
[redacted]:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> reduce(lambda a,b: a+b, range(3))
3
>>> reduce(lambda a,b: a+b, range(4))
6
>>> reduce(lambda a,b: a+b, range(5))
10
>>> reduce(lambda a,b: a*b, range(5))
0
>>> reduce(lambda a,b: a*b, range(1,5))
24
>>>
```

12 Reductions

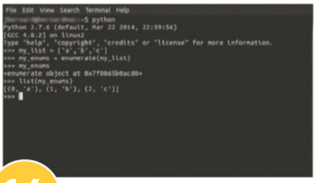
In many calculations, one of the computations you need to do is a reduction operation. This is where you take some list of values and reduce it down to a single value. In Python, you can use the command 'reduce(function, iterable)' to apply the reduction function to each pair of elements in the list. For example, if you apply the summation reduction operation to the list of the first five integers, you would get the result ((((1+2)+3)+4)+5). You can optionally add a third parameter to act as an initialisation term. It is loaded before any elements from the iterable, and is returned as a default if the iterable is actually empty. You can use a lambda function as the function parameter to reduce to keep your code as tight as possible. In this case, remember that it should only take two input parameters.



```
File  Edit  View  Search  Terminal  Help
[redacted]:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> my_bools = [True, True, False, False]
>>> all(my_bools)
False
>>> any(my_bools)
True
>>> my_list = [0,1,2,3]
>>> all(my_list)
False
>>> any(my_list)
True
>>> my_list2 = ['a', 'b', 'c']
>>> all(my_list2)
True
>>> any(my_list2)
True
>>> 
```

13 How true is a list?

In some cases, you may have collected a number of elements within a list that can be evaluated to True or False. For example, maybe you ran a number of possibilities through your computation and have created a list of which ones passed. You can use the command 'any(list)' to check to see whether any of the elements within your list are true. If you need to check whether all of the elements are True, you can use the command 'all(list)'. Both of these commands return a True if the relevant condition is satisfied, and a False if not. They do behave differently if the iterable object is empty, however. The command 'all' returns a True if the iterable is empty, whereas the command 'any' returns a False when given any empty iterable.



```
File  Edit  View  Search  Terminal  Help
[redacted]:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> my_list = ['a','b','c']
>>> my_enums = enumerate(my_list)
>>> my_enums
<enumerate object at 0x7f0865b0acd0>
>>> list(my_enums)
[(0, 'a'), (1, 'b'), (2, 'c')]
>>> 
```

14 Enumerating

Sometimes, we need to label the elements that reside within an iterable object with their indices so that they can be processed at some later point. You could do this by explicitly looping through each of the elements and building an enumerated list. The enumerate command does this in one line. It takes an iterable object and creates a list of tuples as the result. Each tuple has the 0-based index of the element, along with the element itself. You can optionally start the indexing from some other value by including an optional second parameter. As an example, you could enumerate a list of names with the command 'list(enumerate(names, start=1))'. In this example, we decided to start the indexing at 1 instead of 0.

15 Casting

Variables in Python don't have any type information, and so can be used to store any type of object. The actual data, however, is of one type or another. Many operators, like addition, assume that the input values are of the same type. Very often, the operator you are using is smart enough to make the type of conversion that is needed. If you have the need to explicitly convert your data from one type to another, there are a class of functions that can be used to do this conversion process. The ones you are most likely to use is 'abs', 'bin', 'bool', 'chr', 'complex', 'float', 'hex', 'int', 'long', 'oct', and 'str'. For the number-based conversion functions, there is an order of precedence where some types are a subset of others. For example, integers are "lower" than floats. When converting up, no changes in the ultimate value should happen. When converting down, usually some amount of information is lost. For example, when converting from float to integer, Python truncates the number towards zero.

16 What is this?

Everything in Python is an object. You can check to see what class this object is an instance of with the command 'isinstance(object, class)'. This command returns a Boolean value.

17 Is it a subclass?

The command 'issubclass(class1, class2)' checks to see if class1 is a subclass of class2. If class1 and class2 are the same, this is returned as True.

18 Global objects

You can get a dictionary of the global symbol table for the current module with the command 'globals()'.

19 Local objects

You can access an updated dictionary of the current local symbol table by using the command 'locals()'.

20 Variables

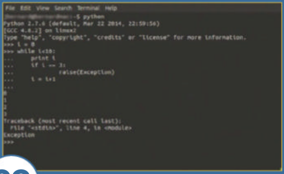
The command 'vars(dict)' returns writeable elements for an object. If you use 'vars()', it behaves like 'locals()'.

21 Making a global

A list of names can be interpreted as globals for the entire code block with the command 'global names'.

22 Nonlocals

In Python 3.X, you can access names from the nearest enclosing scope with the command 'nonlocal names' and bind it to the local scope.



```
File  Edit  View  Search  Terminal  Help
[redacted]:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> i = 0
>>> while i<10:
...     print i
...     if i == 3:
...             raise(Exception)
...     i = i+1
...
0
1
2
3
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
Exception
>>>
```

23 Raising an exception

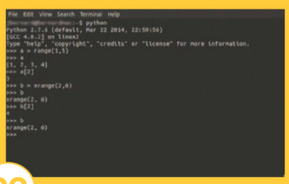
When you identify an error condition, you can use the 'raise' command to throw up an exception. You can include an exception type and a value.

24 Dealing with an exception

Exceptions can be caught in a try-except construction. If the code in the try block raises an exception, the code in the except block gets run.

25 Static methods

You can create a statis method, similar to that in Java or C++, with the command 'staticmethod(function_name)'.



```
[redacted]:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> a = range(1,5)
>>> a
[1, 2, 3, 4]
>>> a[2]
3
>>> b = xrange(2,6)
>>> b
xrange(2, 6)
>>> b[2]
4
>>> b
xrange(2, 6)
>>>
```

26 Ranges

You may need a list of numbers, maybe in a 'for' loop. The command 'range()' can create an iterable list of integers. With one parameter, it goes from 0 to the given number. You can provide an optional start number, as well as a step size. Negative numbers count down.

27 Xranges

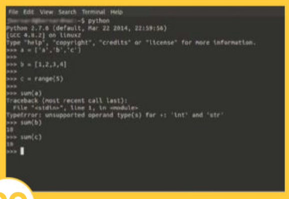
One problem with ranges is that all of the elements need to be calculated up front and stored in memory. The command 'xrange()' takes the same parameters and provides the same result, but only calculates the next element as it is needed.

28 Iterators

Iteration is a very Pythonic way of doing things. For objects which are not intrinsically iterable, you can use the command 'iter(object_name)' to essentially wrap your object and provide an iterable interface for use with other functions and operators.

29 Sorted lists

You can use the command 'sorted(list1)' to sort the elements of a list. You can give it a custom comparison function, and for more complex elements you can include a key function that pulls out a ranking property from each element for comparison.



```
[redacted]:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> a = ['a','b','c']
>>>
>>> b = [1,2,3,4]
>>>
>>> c = range(5)
>>>
>>> sum(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> sum(b)
10
>>> sum(c)
10
>>> █
```

30 Summing items

Above, we saw the general reduction function reduce. A specific type of reduction operation, summation, is common enough to warrant the inclusion of a special case, the command 'sum(iterable_object)'. You can include a second parameter here that will provide a starting value.

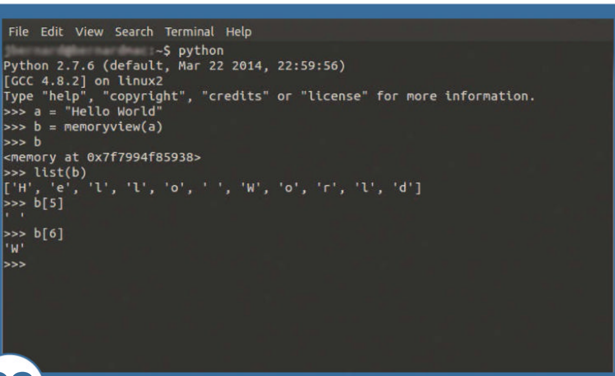
31 With modules

The 'with' command provides the ability to wrap a code block with methods defined by a context manager. This can help clean up code and make it easier to read what a given piece of code is supposed to be doing months later. A classic example of using 'with' is when dealing with files. You could use something like 'with open("myfile.txt", "r") as f:'. This will open the file and prepare it for reading. You can then read the file in the code block with 'data=f.read()'. The best part of doing this is that the file will automatically be closed when the code block is exited, regardless of the reason. So, even if the code block throws an exception, you don't need to worry about closing the file as part of your exception handler. If you have a more complicated 'with' example, you can create a context manager class to help out.

32 Printing

The most direct way of getting output to the user is with the print command. This will send text out to the console window. If you are using version 2.X of Python, there are a couple of ways you can use the print command. The most common way had been simply call it as 'print "Some text"'. You can also use print with the same syntax that you would use for any other function. So, the above example would look like 'print("Some text")'. This is the only form available in version 3.X. If you use the function syntax, you can add extra parameters that give you finer control over this output. For example, you can give the parameter 'file=myfile.txt' and get the output from the print command being dumped into the given text file. It also will accept any object that has some string representation available.

"A classic example of using 'with' is when dealing with files. The best part of doing this is that the file will automatically be closed when the code block is exited, regardless of the reason"



```
[redacted]:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> a = "Hello World"
>>> b = memoryview(a)
>>> b
<memory at 0x7f7994f85938>
>>> list(b)
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
>>> b[5]
' '
>>> b[6]
'W'
>>>
```

33 Memoryview

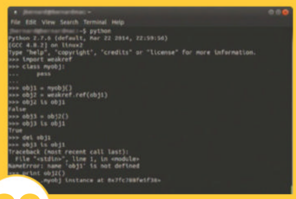
Sometimes, you need to access the raw data of some object, usually as a buffer of bytes. You can copy this data and put it into a bytearray, for example. But this means that you will be using extra memory, and this might not be an option for large objects. The command 'memoryview(object_name)' wraps the object handed in to the command and provides an interface to the raw bytes. It gives access to these bytes an element at a time. In many cases, elements are the size of one byte. But, depending on the object details, you could end up with elements that are larger than that. You can find out the size of an element in bytes with the property 'itemsize'. Once you have your memory view created, you can access the individual elements as you would get elements from a list (mem_view[1], for example).

34 Files

When dealing with files, you need to create a file object to interact with it. The file command takes a string with the file name and location and creates a file object instance. You can then call the file object methods like 'open', 'read' and 'close', to get data out of the file. If you are doing file processing, you can also use the 'readline' method. When opening a file, there is an explicit 'open()' command to simplify the process. It takes a string with the file name, and an optional parameter that is a string which defines the mode. The default is to open the file as read-only ('r'). You can also open it for writing ('w') and appending ('a'). After opening the file, a file object is returned so that you can further interact with it. You can then read it, write to it, and finally close it.

35 Yielding

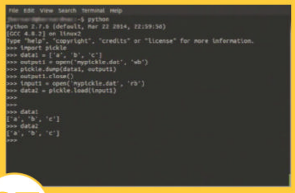
In many cases, a function may need to yield the context of execution to some other function. This is the case with generators. The preferred method for a generator is that it will only calculate the next value when it is requested through the method 'next()'. The command 'yield' saves the current state of the generator function, and return execution control to the calling function. In this way, the saved state of the generator is reloaded and the generator picks up where it left off in order to calculate the next requested value. In this way, you only need to have enough memory available to store the bare minimum to calculate the next needed value, rather than having to store all of the possible values in memory all at once.



```
File  Edit  View  Search  Terminal  Help
[redacted]:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import weakref
>>> class myobj:
...     pass
...
>>> obj1 = myobj()
>>> obj2 = weakref.ref(obj1)
>>> obj2 is obj1
False
>>> obj3 = obj2()
>>> obj3 is obj1
True
>>> del obj1
>>> obj3 is obj1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'obj1' is not defined
>>> print obj2()
<__main__.myobj instance at 0x7fc788fe1f38>
```

36 Weak references

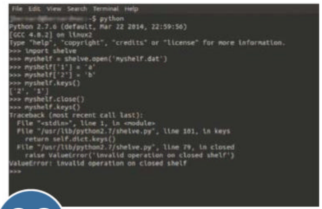
You sometimes need to have a reference to an object, but still be able to destroy it if needed. A weak reference is one which can be ignored by the garbage collector. If the only references left to n object are weak references, then the garbage collector is allowed to destroy that object and reclaim the space for other uses. This is useful in cases where you have caches or mappings of large datasets that don't necessarily have to stay in memory. If an object that is weakly referenced ends up being destroyed and you try to access it, it will appear as a None. You can test for this condition and then reload the data if you decide that this is a necessary step.



```
File  Edit  View  Search  Terminal  Help
[redacted]:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import pickle
>>> data1 = ['a', 'b', 'c']
>>> output1 = open('mypickle.dat', 'wb')
>>> pickle.dump(data1, output1)
>>> output1.close()
>>> input1 = open('mypickle.dat', 'rb')
>>> data2 = pickle.load(input1)
>>>
>>>
>>> data1
['a', 'b', 'c']
>>> data2
['a', 'b', 'c']
>>>
```

37 Pickling data

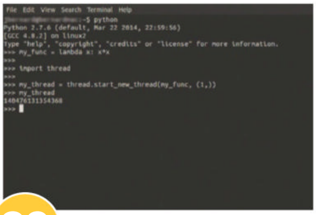
There are a few different ways of serialising memory when you need to checkpoint results to disk. One of these is called pickling. Pickle is actually a complete module, not just a single command. To store data on to the hard drive, you can use the dump method to write the data out. When you want to reload the same data at some other point in the future, you can use the load method to read the data in and unpickle it. One issue with pickle is its speed, or lack of it. There is a second module, cPickle, that provides the same basic functionality. But, since it is written in C, it can be as much as 1000 times faster. One thing to be aware of is that pickle does not store any class information for an object, but only its instance information. This means that when you unpickle the object, it may have different methods and attributes if the class definition has changed in the interim.



```
File  Edit  View  Search  Terminal  Help
[redacted]:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import shelve
>>> myshelf = shelve.open('myshelf.dat')
>>> myshelf['1'] = 'a'
>>> myshelf['2'] = 'b'
>>> myshelf.keys()
['2', '1']
>>> myshelf.close()
>>> myshelf.keys()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python2.7/shelve.py", line 101, in keys
    return self.dict.keys()
  File "/usr/lib/python2.7/shelve.py", line 79, in closed
    raise ValueError('invalid operation on closed shelf')
ValueError: invalid operation on closed shelf
>>>
```

38 Shelving data

While pickling allows you save data and reload it, sometimes you need more structured object permanence in your Python session. With the shelve module, you can create an object store where essentially anything that can be pickled can be stored there. The backend of the storage on the drive can be handled by one of several systems, such as dbm or gdbm. Once you have opened a shelf, you can read and write to it using key value pairs. When you are done, you need to be sure to explicitly close the shelf so that it is synchronised with the file storage. Because of the way the data may be stored in the backing database, it is best to not open the relevant files outside of the shelve module in Python. You can also open the shelf with writeback set to True. If so, you can explicitly call the sync method to write out cached changes.



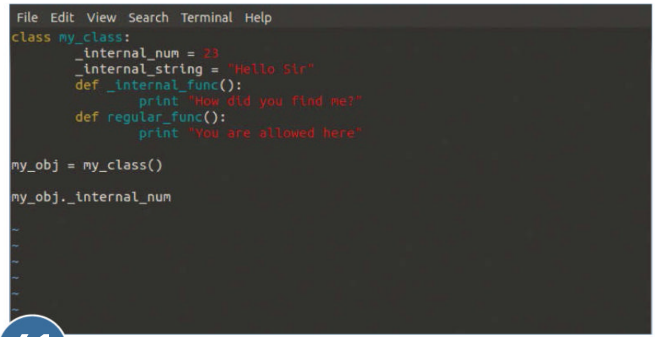
```
File  Edit  View  Search  Terminal  Help
[redacted]:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> my_func = lambda x: x*x
>>>
>>> import thread
>>>
>>> my_thread = thread.start_new_thread(my_func, (1,))
>>> my_thread
140476131354368
>>> 
```

39 Threads

You can do multiple threads of execution within Python. The 'thread()' command can create a new thread of execution for you. It follows the same techniques as those for POSIX threads. When you first create a thread, you need to hand in a function name, along with whatever parameters said function needs. One thing to keep in mind is that these threads behave just like POSIX threads. This means that almost everything is the responsibility of the programmer. You need to handle mutex locks (with the methods 'acquire' and 'release'), as well as create the original mutexes with the method 'allocate_lock'. When you are done, you need to 'exit' the thread to ensure that it is properly cleaned up and no resources get left behind. You also have fine-grained control over the threads, being able to set things like the stack size for new threads.

40 Inputting data

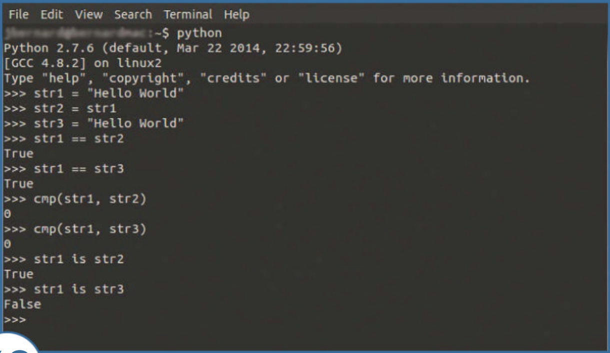
Sometimes, you need to collect input from an end user. The command 'input()' can take a prompt string to display to the user, and then wait for the user to type a response. Once the user is done typing and hits the enter key, the text is returned to your program. If the readline module was loaded before calling input, then you will have enhanced line editing and history functionality. This command passes the text through eval first, and so may cause uncaught errors. If you have any doubts, you can use the command 'raw_input()' to skip this problem. This command simply returns the unchanged string inputted by the user. Again, you can use the readline module to get enhanced line editing.



```
class my_class:
        _internal_num = 23
        _internal_string = "Hello Sir"
        def _internal_func():
                print "How did you find me?"
        def regular_func():
                print "You are allowed here"

my_obj = my_class()

my_obj._internal_num

~
~
~
~
~
```

41 Internal variables

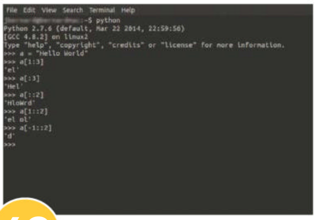
For people coming from other programming languages, there is a concept of having certain variables or methods be only available internally within an object. In Python, there is no such concept. All elements of an object are accessible. There is a style rule, however, that can mimic this type of behaviour. Any names that start with an underscore are expected to be treated as if they were internal names and to be kept as private to the object. They are not hidden, however, and there is no explicit protection for these variables or methods. It is up to the programmer to honour the intention from the author the class and not alter any of these internal names. You are free to make these types of changes if it becomes necessary, though.



```
[redacted]:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> str1 = "Hello World"
>>> str2 = str1
>>> str3 = "Hello World"
>>> str1 == str2
True
>>> str1 == str3
True
>>> cmp(str1, str2)
0
>>> cmp(str1, str3)
0
>>> str1 is str2
True
>>> str1 is str3
False
>>>
```

42 Comparing objects

There are several ways to compare objects within Python, with several caveats. The first is that you can test two things between objects: equality and identity. If you are testing identity, you are testing to see if two names actually refer to the same instance object. This can be done with the command 'cmp(obj1, obj2)'. You can also test this condition by using the 'is' keyword. For example, 'obj1 is obj2'. If you are testing for equality, you are testing to see whether the values in the objects referred to by the two names are equal. This test is handled by the operator '==', as in 'obj1 == obj2'. Testing for equality can become complex for more complicated objects.

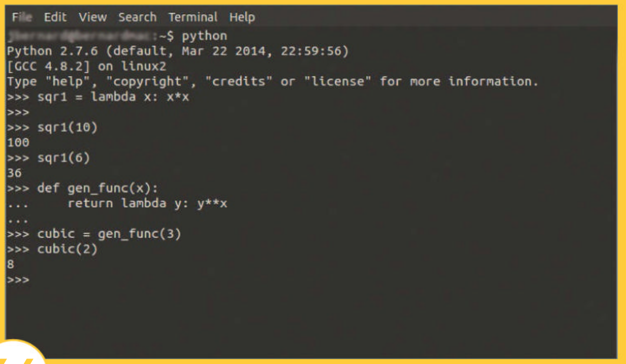


```
[redacted]:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> a = "Hello World"
>>> a[1:3]
'el'
>>> a[:3]
'Hel'
>>> a[::2]
'HloWrd'
>>> a[1::2]
'el ol'
>>> a[-1::2]
'd'
>>>
```

43 Slices

While not truly a command, slices are too important a concept not to mention in this list of essential commands. Indexing elements in data structures, like lists, is one of the most common things done in Python. You can select a single element by giving a single index value. More interestingly, you can select a range of elements by giving a start index and an end index, separated by a colon. This gets returned as a new list that you can save in a new variable name. You can even change the step size, allowing you to skip some number of elements. So, you could grab every odd element from the list 'a' with the slice 'a[1::2]'. This starts at index 1, continues until the end, and steps through the index values 2 at a time. Slices can be given negative index values. If you do, then they start from the end of the list and count backwards.

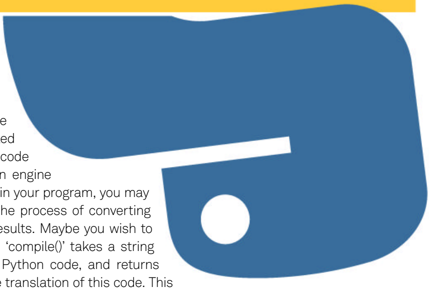
"Python is an interpreted language, which means that the source code that you write needs to be compiled into a byte code format. This byte code then gets fed into the actual Python engine"



```
File  Edit  View  Search  Terminal  Help
[redacted]:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> sqr1 = lambda x: x*x
>>>
>>> sqr1(10)
100
>>> sqr1(6)
36
>>> def gen_func(x):
...     return lambda y: y**x
...
>>> cubic = gen_func(3)
>>> cubic(2)
8
>>>
```

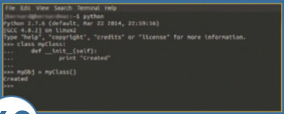
44 Lambda expressions

Since objects, and the names that point to them, are truly different things, you can have objects that have no references to them. One example of this is the lambda expression. With this, you can create an anonymous function. This allows you use functional programming techniques within Python. The format is the keyword 'lambda', followed by a parameter list, then a colon and the function code. For example, you could build your own function to square a number with 'lambda x: x*x'. You can then have a function that can programmatically create new functions and return them to the calling code. With this capability, you can create function generators to have self-modifying programs. The only limitation is that they are limited to a single expression, so you can't generate very complex functions.



45 Compiling code objects

Python is an interpreted language, which means that the source code that you write needs to be compiled into a byte code format. This byte code then gets fed into the actual Python engine to step through the instructions. Within your program, you may have the need to take control over the process of converting code to byte code and running the results. Maybe you wish to build your own REPL. The command 'compile()' takes a string object that contains a collection of Python code, and returns an object that represents a byte code translation of this code. This new object can then be handed in to either 'eval()' or 'exec()' to be actually run. You can use the parameter 'mode=' to tell compile what kind of code is being compiled. The 'single' mode is a single statement, 'eval' is a single expression and 'exec' is a whole code block.



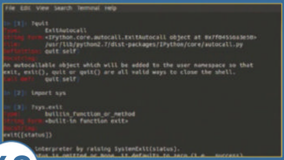
```
File  Edit  View  Search  Terminal  Help
[redacted]:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> class myClass:
...     def __init__(self):
...         print "Created"
...
>>> myObj = myClass()
Created
>>>
```

46 __init__ method

When you create a new class, you can include a private initialisation method that gets called when a new instance of the class is created. This method is useful when the new object instance needs some data loaded in the new object.

47 __del__ method

When an instance object is about to be destroyed, the __del__ method is called. This gives you the chance to do any kind of cleanup that may be required. This might be closing files, or disconnecting network connections. After this code is completed, the object is finally destroyed and resources are freed.



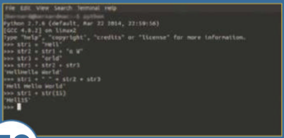
```
File  Edit  View  Search  Terminal  Help

In [1]: ?quit
Type:        ExitAutocall
String form: <IPython.core.autocall.ExitAutocall object at 0x7f04550a3e50>
File:        /usr/lib/python2.7/dist-packages/IPython/core/autocall.py
Definition:  quit(self)
Docstring:
An autocallable object which will be added to the user namespace so that
exit, exit(), quit or quit() are all valid ways to close the shell.
Call def:    quit(self)

In [2]: import sys

In [3]: ?sys.exit
Type:        builtin_function_or_method
String form: <built-in function exit>
Docstring:
exit([status])

         interpreter by raising SystemExit(status).
```

48 Exiting your program

There are two pseudo-commands available to exit from the Python interpreter: 'exit()' and quit()'. They both take an optional parameter which sets the exit code for the process. If you want to exit from a script, you are better off using the exit function from the sys module ('sys.exit(exit_code)'.

49 Return values

Functions may need to return some value to the calling function. Because essentially no name has a type, this includes functions. So functions can use the 'return' command to return any object to the caller.



```
File  Edit  View  Search  Terminal  Help
[redacted]:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> str1 = "Hell"
>>> str2 = str1 + "o W"
>>> str3 = "orld"
>>> str1 + str2 + str3
'HellHello World'
>>> str1 + " " + str2 + str3
'Hell Hello World'
>>> str1 + str(15)
'Hell15'
>>> █
```

50 String concatenation

We will finish with what most lists start with – string concatenation. The easiest way to build up strings is to use the '+' operator. If you want to include other items, like numbers, you can use the 'str()' casting function to convert it to a string object.