

Working with Data



Data is everything. It's more valuable than gold or oil. With data, governments can change the world, politicians can rise or fall and companies, large or small, can impact our future in ways we wouldn't have imagined just a few years ago. Data is power, and learning how to manipulate and control it are essential aspects of any coding language.

We take a look at how you can use date and time functions, write to files in your system and even create graphical user interfaces that will take your coding skills to new levels and open more doors for you.

-
- 66** Lists
 - 68** Tuples
 - 70** Dictionaries
 - 72** Splitting and Joining Strings
 - 74** Formatting Strings
 - 76** Date and Time
 - 78** Opening Files
 - 80** Writing to Files
 - 82** Exceptions
 - 84** Python Graphics
 - 86** Combining What You Know So Far



Lists

Lists are one of the most common types of data structures you will come across in Python. A list is simply a collection of items, or data if you prefer, that can be accessed as a whole, or individually if wanted.

WORKING WITH LISTS

Lists are extremely handy in Python. A list can be strings, integers and also variables. You can even include functions in lists, and lists within lists.

STEP 1

A list is a sequence of data values called items. You create the name of your list followed by an equals sign, then square brackets and the items separated by commas; note that strings use quotes:

```
numbers = [1, 4, 7, 21, 98, 156]
mythical_creatures = ["Unicorn", "Balrog",
"Vampire", "Dragon", "Minotaur"]
```

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> numbers = [1, 4, 7, 21, 98, 156]
>>> mythical_creatures = ["Unicorn", "Balrog", "Vampire", "Dragon", "Minotaur"]
>>>
```

STEP 2

Once you've defined your list you can call each by referencing its name, followed by a number. Lists start the first item entry as 0, followed by 1, 2, 3 and so on. For example:

`numbers`

To call up the entire contents of the list.

`numbers[3]`

To call the third from zero item in the list (21 in this case).

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> numbers = [1, 4, 7, 21, 98, 156]
>>> mythical_creatures = ["Unicorn", "Balrog", "Vampire", "Dragon", "Minotaur"]
>>> numbers
[1, 4, 7, 21, 98, 156]
>>> numbers[3]
21
>>> mythical_creatures
['Unicorn', 'Balrog', 'Vampire', 'Dragon', 'Minotaur']
>>> mythical_creatures[3]
'Dragon'
>>>
```

STEP 3

You can also access, or index, the last item in a list by using the minus sign before the item number [-1], or the second to last item with [-2] and so on. Trying to reference an item that isn't in the list, such as [10] will return an error:

```
numbers[-1]
mythical_creatures[-4]
```

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> numbers = [1, 4, 7, 21, 98, 156]
>>> mythical_creatures = ["Unicorn", "Balrog", "Vampire", "Dragon", "Minotaur"]
>>> numbers
[1, 4, 7, 21, 98, 156]
>>> numbers[3]
21
>>> mythical_creatures
['Unicorn', 'Balrog', 'Vampire', 'Dragon', 'Minotaur']
>>> mythical_creatures[3]
'Dragon'
>>> numbers[-1]
156
>>> numbers[-2]
98
>>> mythical_creatures[-1]
'Minotaur'
>>> mythical_creatures[-4]
'Balrog'
>>>
```

STEP 4

Slicing is similar to indexing but you can retrieve multiple items in a list by separating item numbers with a colon. For example:

`numbers[1:3]`

Will output the 4 and 7, being item numbers 1 and 2. Note that the returned values don't include the second index position (as you would `numbers[1:3]` to return 4, 7 and 21).

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> numbers = [1, 4, 7, 21, 98, 156]
>>> mythical_creatures = ["Unicorn", "Balrog", "Vampire", "Dragon", "Minotaur"]
>>> numbers
[1, 4, 7, 21, 98, 156]
>>> numbers[3]
21
>>> mythical_creatures
['Unicorn', 'Balrog', 'Vampire', 'Dragon', 'Minotaur']
>>> mythical_creatures[3]
'Dragon'
>>> numbers[-1]
156
>>> numbers[-2]
98
>>> mythical_creatures[-1]
'Minotaur'
>>> mythical_creatures[-4]
'Balrog'
>>> numbers[1:3]
[4, 7]
>>> numbers[0:4]
[1, 4, 7, 21]
>>> numbers[3:5]
[21, 98]
>>> numbers[1:]
[4, 7, 21, 98, 156]
>>>
```

STEP 5 You can update items within an existing list, remove items and even join lists together. For example, to join two lists you can use:

```
everything = numbers + mythical_creatures
```

Then view the combined list with:

```
everything
```

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> numbers = [1, 4, 7, 21, 98, 156]
>>> mythical_creatures = ["Unicorn", "Balrog", "Vampire", "Dragon", "Minotaur"]
>>> everything = numbers + mythical_creatures
>>> everything
[1, 4, 7, 21, 98, 156, 'Unicorn', 'Balrog', 'Vampire', 'Dragon', 'Minotaur']
```

STEP 6 Items can be added to a list by entering:

```
numbers=numbers+[201]
```

Or for strings:

```
mythical_creatures=mythical_creatures+["Griffin"]
```

Or by using the append function:

```
mythical_creatures.append("Nessie")
numbers.append(278)
```

```
>>> numbers = [1, 4, 7, 21, 98, 156]
>>> mythical_creatures = ["Unicorn", "Balrog", "Vampire", "Dragon", "Minotaur"]
>>> numbers
[1, 4, 7, 21, 98, 156]
>>> mythical_creatures
['Unicorn', 'Balrog', 'Vampire', 'Dragon', 'Minotaur']
>>> numbers=numbers+[201]
>>> numbers
[1, 4, 7, 21, 98, 156, 201]
>>> mythical_creatures=mythical_creatures+["Griffin"]
>>> mythical_creatures
['Unicorn', 'Balrog', 'Vampire', 'Dragon', 'Minotaur', 'Griffin']
>>> mythical_creatures.append("Nessie")
>>> mythical_creatures
['Unicorn', 'Balrog', 'Vampire', 'Dragon', 'Minotaur', 'Griffin', 'Nessie']
>>> numbers.append(278)
>>> numbers
[1, 4, 7, 21, 98, 156, 201, 278]
```

STEP 7 Removal of items can be done in two ways. The first is by the item number:

```
del numbers[7]
```

Alternatively, by item name:

```
mythical_creatures.remove("Nessie")
```

```
>>> mythical_creatures = ["Unicorn", "Balrog", "Vampire", "Dragon", "Minotaur"]
>>> numbers
[1, 4, 7, 21, 98, 156]
>>> mythical_creatures
['Unicorn', 'Balrog', 'Vampire', 'Dragon', 'Minotaur']
>>> numbers=numbers+[201]
>>> numbers
[1, 4, 7, 21, 98, 156, 201]
>>> mythical_creatures=mythical_creatures+["Griffin"]
>>> mythical_creatures
['Unicorn', 'Balrog', 'Vampire', 'Dragon', 'Minotaur', 'Griffin']
>>> mythical_creatures.append("Nessie")
>>> mythical_creatures
['Unicorn', 'Balrog', 'Vampire', 'Dragon', 'Minotaur', 'Griffin', 'Nessie']
>>> numbers.append(278)
>>> del numbers[7]
>>> numbers
[1, 4, 7, 21, 98, 156, 201]
>>> mythical_creatures.remove("Nessie")
>>> mythical_creatures
['Unicorn', 'Balrog', 'Vampire', 'Dragon', 'Minotaur', 'Griffin']
```

STEP 8 You can view what can be done with lists by entering dir(list) into the Shell. The output is the available functions, for example, insert and pop are used to add and remove items at certain positions. To insert the number 62 at item index 4:

```
numbers.insert(4, 62)
```

To remove it:

```
numbers.pop(4)
```

```
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattribute__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> numbers = [1, 4, 7, 21, 98, 156]
>>> numbers
[1, 4, 7, 21, 98, 156]
>>> numbers.insert(4, 62)
>>> numbers
[1, 4, 7, 21, 62, 98, 156]
#
>>> numbers.pop(4)
62
>>> numbers
[1, 4, 7, 21, 98, 156]
>>>
```

STEP 9 You also use the list function to break a string down into its components. For example:

```
list("David")
```

Breaks the name David into 'D', 'a', 'v', 'i', 'd'. This can then be passed to a new list:

```
name=list("David Hayward")
name
age=[44]
user = name + age
user
```

```
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> list("David")
['D', 'a', 'v', 'i', 'd']
>>> name=list("David Hayward")
>>> name
['D', 'a', 'v', 'i', 'd', ' ', 'H', 'a', 'y', 'w', 'a', 'r', 'd']
>>> age=[44]
>>> user = name + age
>>> user
['D', 'a', 'v', 'i', 'd', ' ', 'H', 'a', 'y', 'w', 'a', 'r', 'd', 44]
>>>
```

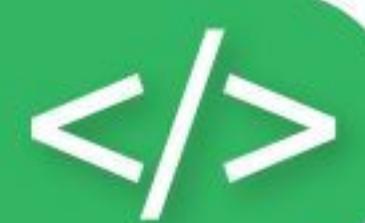
STEP 10 Based on that, you can create a program to store someone's name and age as a list:

```
name=input("What's your name? ")
lname=list(name)
age=int(input("How old are you: "))
lage=[age]
```

```
user = lname + lage
```

The combined name and age list is called user, which can be called by entering user into the Shell. Experiment and see what you can do.

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> -----
>>> What's your name? Conan of Cimmeria
>>> How old are you: 44
>>> user
['C', 'o', 'n', 'a', ' ', 'o', 'f', ' ', 'C', 'i', 'm', 'e', 'r', 'i', 'a', ' ', '4', '4']
>>>
```



Tuples

Tuples are very much identical to lists. However, where lists can be updated, deleted or changed in some way, a tuple remains a constant. This is called immutable and they're perfect for storing fixed data items.

THE IMMUTABLE TUPLE

Reasons for having tuples vary depending on what the program is intended to do. Normally, a tuple is reserved for something special but they're also used for example, in an adventure game, where non-playing character names are stored.

STEP 1

A tuple is created the same way as a list but in this instance you use curved brackets instead of square brackets. For example:

```
months=("January", "February", "March", "April",
"May", "June")
months
```

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> months=("January", "February", "March", "April",
>>> "May", "June")
>>> months
('January', 'February', 'March', 'April', 'May', 'June')
>>> |
```

STEP 2

Just as with lists, the items within a named tuple can be indexed according to their position in the data range, i.e.:

```
months[0]
months[5]
```

However, any attempt at deleting or adding to the tuple will result in an error in the Shell.

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> months=("January", "February", "March", "April", "May", "June")
>>> months
('January', 'February', 'March', 'April', 'May', 'June')
>>> months[0]
'January'
>>> months[5]
'June'
>>> months.append("July")
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    months.append("July")
AttributeError: 'tuple' object has no attribute 'append'
>>> |
```

STEP 3

You can create grouped tuples into lists that contain multiple sets of data. For instance, here is a tuple called NPC (Non-Playable Characters) containing the character name and their combat rating for an adventure game:

```
NPC=[("Conan", 100), ("Belit", 80),
("Valeria", 95)]
```

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> NPC=[("Conan",100), ("Belit", 80), ("Valeria", 95)]
>>> |
>>> |
```

STEP 4

Each of these data items can be accessed as a whole by entering NPC into the Shell; or they can be indexed according to their position NPC[0]. You can also index the individual tuples within the NPC list:

```
NPC[0][1]
```

Will display 100.

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> NPC=[("Conan", 100), ("Belit", 80), ("Valeria", 95)]
>>> NPC
[('Conan', 100), ('Belit', 80), ('Valeria', 95)]
>>> NPC[0]
('Conan', 100)
>>> NPC[0][1]
100
>>> |
>>> |
```

STEP 5

It's worth noting that when referencing multiple tuples within a list, the indexing is slightly different from the norm. You would expect the 95 combat rating of the character Valeria to be NPC[4][5], but it's not. It's actually:

`NPC[2][1]`

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> NPC=[("Conan", 100), ("Belit", 80), ("Valeria", 95)]
>>> NPC[2][1]
95
>>> |
```

STEP 6

This means of course that the indexing follows thus:

0	1, 1
0, 0	2
0, 1	2, 0
1	2, 1
1, 0	

Which as you can imagine, gets a little confusing when you've got a lot of tuple data to deal with.

```
>>> NPC=[("Conan", 100), ("Belit", 80), ("Valeria", 95)]
>>> NPC[0]
('Conan', 100)
>>> NPC[0][0]
'Conan'
>>> NPC[0][1]
100
>>> NPC[1]
('Belit', 80)
>>> NPC[1][0]
'Belit'
>>> NPC[1][1]
80
>>> NPC[2]
('Valeria', 95)
>>> NPC[2][0]
'Valeria'
>>> NPC[2][1]
95
>>> |
```

STEP 7

Tuples though utilise a feature called unpacking, where the data items stored within a tuple are assigned variables. First create the tuple with two items (name and combat rating):

`NPC=("Conan", 100)`

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> NPC=("Conan", 100)
>>> |
```

STEP 8

Now unpack the tuple into two corresponding variables:

`(name, combat_rating)=NPC`

You can now check the values by entering name and combat_rating.

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> NPC=("Conan", 100)
>>> (name, combat_rating)=NPC
>>> name
'Conan'
>>> combat_rating
100
>>> |
```

STEP 9

Remember, as with lists, you can also index tuples using negative numbers which count backwards from the end of the data list. For our example, using the tuple with multiple data items, you would reference the Valeria character with:

`NPC[2][-0]`

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> NPC=[("Conan", 100), ("Belit", 80), ("Valeria", 95)]
>>> NPC[2][-0]
'Valeria'
>>> |
```

STEP 10

You can use the max and min functions to find the highest and lowest values of a tuple composed of numbers. For example:

`numbers=(10.3, 23, 45.2, 109.3, 6.1, 56.7, 99)`

The numbers can be integers and floats. To output the highest and lowest, use:

```
print(max(numbers))
print(min(numbers))
```

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> numbers=(10.3, 23, 45.2, 109.3, 6.1, 56.7, 99)
>>> print(max(numbers))
109.3
>>> print(min(numbers))
6.1
>>> |
```



Dictionaries

Lists are extremely useful but dictionaries in Python are by far the more technical way of dealing with data items. They can be tricky to get to grips with at first but you'll soon be able to apply them to your own code.

KEY PAIRS

A dictionary is like a list but instead each data item comes as a pair, these are known as Key and Value. The Key part must be unique and can either be a number or string whereas the Value can be any data item you like.

STEP 1

Let's say you want to create a phonebook in Python. You would create the dictionary name and enter the data in curly brackets, separating the key and value by a colon **Key:Value**. For example:

```
phonebook={"Emma": 1234, "Daniel": 3456,  
"Hannah": 6789}
```

The screenshot shows the Python 3.4.2 Shell window. The command `phonebook={"Emma": 1234, "Daniel": 3456, "Hannah": 6789}` is entered and executed, resulting in the output: `Python 3.4.2 (default, Oct 19 2014, 13:31:11) [GCC 4.9.1] on linux Type "copyright", "credits" or "license()" for more information.`

STEP 3

As with lists and tuples, you can check the contents of a dictionary by giving the dictionary a name: `phonebook`, in this example. This will display the data items you've entered in a similar fashion to a list, which you're no doubt familiar with by now.

The screenshot shows the Python 3.4.2 Shell window. The command `print(phonebook)` is entered and executed, resulting in the output: `Python 3.4.2 (default, Oct 19 2014, 13:31:11) [GCC 4.9.1] on linux Type "copyright", "credits" or "license()" for more information.`

STEP 2

Just as with most lists, tuples and so on, strings need be enclosed in quotes (single or double), whilst integers can be left open. Remember that the value can be either a string or an integer, you just need to enclose the relevant one in quotes:

```
phonebook2={"David": "0987 654 321"}
```

The screenshot shows the Python 3.4.2 Shell window. The command `phonebook2={"David": "0987 654 321"}` is entered and executed, resulting in the output: `Python 3.4.2 (default, Oct 19 2014, 13:31:11) [GCC 4.9.1] on linux Type "copyright", "credits" or "license()" for more information.`

STEP 4

The benefit of using a dictionary is that you can enter the key to index the value. Using the phonebook example from the previous steps, you can enter:

```
phonebook["Emma"]  
phonebook["Hannah"]
```

The screenshot shows the Python 3.4.2 Shell window. The command `phonebook["Emma"]` is entered and executed, resulting in the output: `1234`.

STEP 5

Adding to a dictionary is easy too. You can include a new data item entry by adding the new key and value items like:

```
phonebook["David"] = "0987 654 321"
phonebook
```

A screenshot of the Python 3.4.2 Shell window. The command `phonebook["David"] = "0987 654 321"` is entered, followed by `phonebook`. The output shows the updated dictionary: `{'Hannah': 6789, 'Emma': 1234, 'Daniel': 3456, 'David': '0987 654 321'}`.

STEP 6

You can also remove items from a dictionary by issuing the `del` command followed by the item's key; the value will be removed as well, since both work as a pair of data items:

```
del phonebook["David"]
```

A screenshot of the Python 3.4.2 Shell window. The command `del phonebook["David"]` is entered after the previous dictionary definition. The output shows the dictionary without the 'David' entry: `{'Hannah': 6789, 'Emma': 1234, 'Daniel': 3456}`.

STEP 7

Taking this a step further, how about creating a piece of code that will ask the user for the dictionary key and value items? Create a new Editor instance and start by coding in a new, blank dictionary:

```
phonebook={}
```

A screenshot showing two windows side-by-side. The left window is the Python 3.4.2 Shell with the command `phonebook={}`. The right window is a code editor titled `*DictIn.py*` containing the code `phonebook={}`.

STEP 8

Next, you need to define the user inputs and variables: one for the person's name, the other for their phone number (let's keep it simple to avoid lengthy Python code):

```
name=input("Enter name: ")
number=int(input("Enter phone number: "))
```

A screenshot of a code editor window titled `*DictIn.py - /home/pi/Documents/Python Code/DictIn.py (3.4.2)*`. It contains the code `phonebook={}`, `name=input("Enter name: ")`, and `number=int(input("Enter phone number: "))`.

STEP 9

Note we've kept the number as an integer instead of a string, even though the value can be both an integer or a string. Now you need to add the user's inputted variables to the newly created blank dictionary. Using the same process as in Step 5, you can enter:

```
phonebook[name] = number
```

A screenshot of a code editor window titled `*DictIn.py - /home/pi/Documents/Python Code/DictIn.py (3.4.2)*`. It contains the code `phonebook={}`, `name=input("Enter name: ")`, `number=int(input("Enter phone number: "))`, and `phonebook[name] = number`.

STEP 10

Now when you save and execute the code, Python will ask for a name and a number. It will then insert those entries into the phonebook dictionary, which you can test by entering into the Shell:

```
phonebook
phonebook["David"]
```

If the number needs to contain spaces you need to make it a string, so remove the `int` part of the input.

A screenshot showing two windows side-by-side. The left window is the Python 3.4.2 Shell with the command `phonebook`, followed by `phonebook["David"]`. The right window is a code editor titled `*DictIn.py - /home/pi/Documents/Python Code/DictIn.py (3.4.2)*` containing the code `phonebook={}`, `name=input("Enter name: ")`, `number=input("Enter phone number: ")`, and `phonebook[name] = [number]`.



Splitting and Joining Strings

When dealing with data in Python, especially from a user's input, you will undoubtedly come across long sets of strings. A useful skill to learn in Python programming is being able to split those long strings for better readability.

STRING THEORIES

You've already looked at some list functions, using `.insert`, `.remove`, and `.pop` but there are also functions that can be applied to strings.

STEP 1

The main tool in the string function arsenal is `.split()`.

With it you're able to split apart a string of data, based on the argument within the brackets. For example, here's a string with three items, each separated by a space:

```
text="Daniel Hannah Emma"
```

A screenshot of the Python 3.4.2 Shell window. The title bar says "Python 3.4.2 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The command line shows the Python version and path, followed by the assignment of the variable `text` to the string "Daniel Hannah Emma".

STEP 2

Now let's turn the string into a list and split the content accordingly:

```
names=text.split(" ")
```

Then enter the name of the new list, `names`, to see the three items.

A screenshot of the Python 3.4.2 Shell window. It shows the previous code execution followed by the assignment of `names` to the result of `text.split(" ")`. The output is a list containing the three names: ['Daniel', 'Hannah', 'Emma'].

STEP 3

Note that the `text.split` part has the brackets, quotes, then a space followed by closing quotes and brackets. The space is the separator, indicating that each list item entry is separated by a space. Likewise, CSV (Comma Separated Value) content has a comma, so you'd use:

```
text="January,February,March,April,May,June"  
months=text.split(",")  
months
```

A screenshot of the Python 3.4.2 Shell window. It shows the code to split the string "January,February,March,April,May,June" into a list of months. The output is a list: ['January', 'February', 'March', 'April', 'May', 'June'].

STEP 4

You've previously seen how you can split a string into individual letters as a list, using a name:

```
name=list("David")  
name
```

The returned value is 'D', 'a', 'v', 'i', 'd'. Whilst it may seem a little useless under ordinary circumstances, it could be handy for creating a spelling game for example.

A screenshot of the Python 3.4.2 Shell window. It shows the code to convert the string "David" into a list of individual characters. The output is a list: ['D', 'a', 'v', 'i', 'd'].

STEP 5

The opposite of the `.split` function is `.join`, where you will have separate items in a string and can join them all together to form a word or just a combination of items, depending on the program you're writing. For instance:

```
alphabet="" .join(["a", "b", "c", "d", "e"])
alphabet
```

This will display 'abcde' in the Shell.

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> alphabet="" .join(["a", "b", "c", "d", "e"])
>>> alphabet
'abcde'
>>>
```

STEP 6

You can therefore apply `.join` to the separated name you made in Step 4, combining the letters again to form the name:

```
name="" .join(name)
name
```

We've joined the string back together, and retained the list called `name`, passing it through the `.join` function.

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> name=list("David")
>>> name
['D', 'a', 'v', 'i', 'd']
>>> name="" .join(name)
>>> name
'David'
>>>
```

STEP 7

A good example of using the `.join` function is when you have a list of words you want to combine into a sentence:

```
list=["Conan", "raised", "his", "mighty", "sword",
      "and", "struck", "the", "demon"]
text=" ".join(list)
text
```

Note the space between the quotes before the `.join` function (where there were no quotes in Step 6's `.join`).

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> list=["Conan", "raised", "his", "mighty", "sword", "and", "struck", "the", "demon"]
>>> text=" ".join(list)
>>> text
'Conan raised his mighty sword and struck the demon'
>>>
```

STEP 8

As with the `.split` function, the separator doesn't have to be a space, it can also be a comma, a full stop, a hyphen or whatever you like:

```
colours=["Red", "Green", "Blue"]
col="," .join(colours)
col
```

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> list=["Conan", "raised", "his", "mighty", "sword", "and", "struck", "the", "demon"]
>>> text=" ".join(list)
'Conan raised his mighty sword and struck the demon'
>>> colours=["Red", "Green", "Blue"]
>>> col="," .join(colours)
>>> col
'Red,Green,Blue'
```

STEP 9

There's some interesting functions you apply to a string, such as `.capitalize` and `.title`. For example:

```
title="conan the cimmerian"
title.capitalize()
title.title()
```

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> title="conan the cimmerian"
>>> title.capitalize()
'Conan the cimmerian'
>>> title.title()
'Conan The Cimmerian'
>>>
```

STEP 10

You can also use logic operators on strings, with the 'in' and 'not in' functions. These enable you to check if a string contains (or does not contain) a sequence of characters:

```
message="Have a nice day"
"nice" in message
"bad" not in message
"day" not in message
"night" in message
```

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> message="Have a nice day"
>>> "nice" in message
True
>>> "bad" not in message
True
>>> "day" not in message
False
>>> "night" in message
False
>>>
```

Formatting Strings

When you work with data, creating lists, dictionaries and objects you may often want to print out the results. Merging strings with data is easy especially with Python 3, as earlier versions of Python tended to complicate matters.

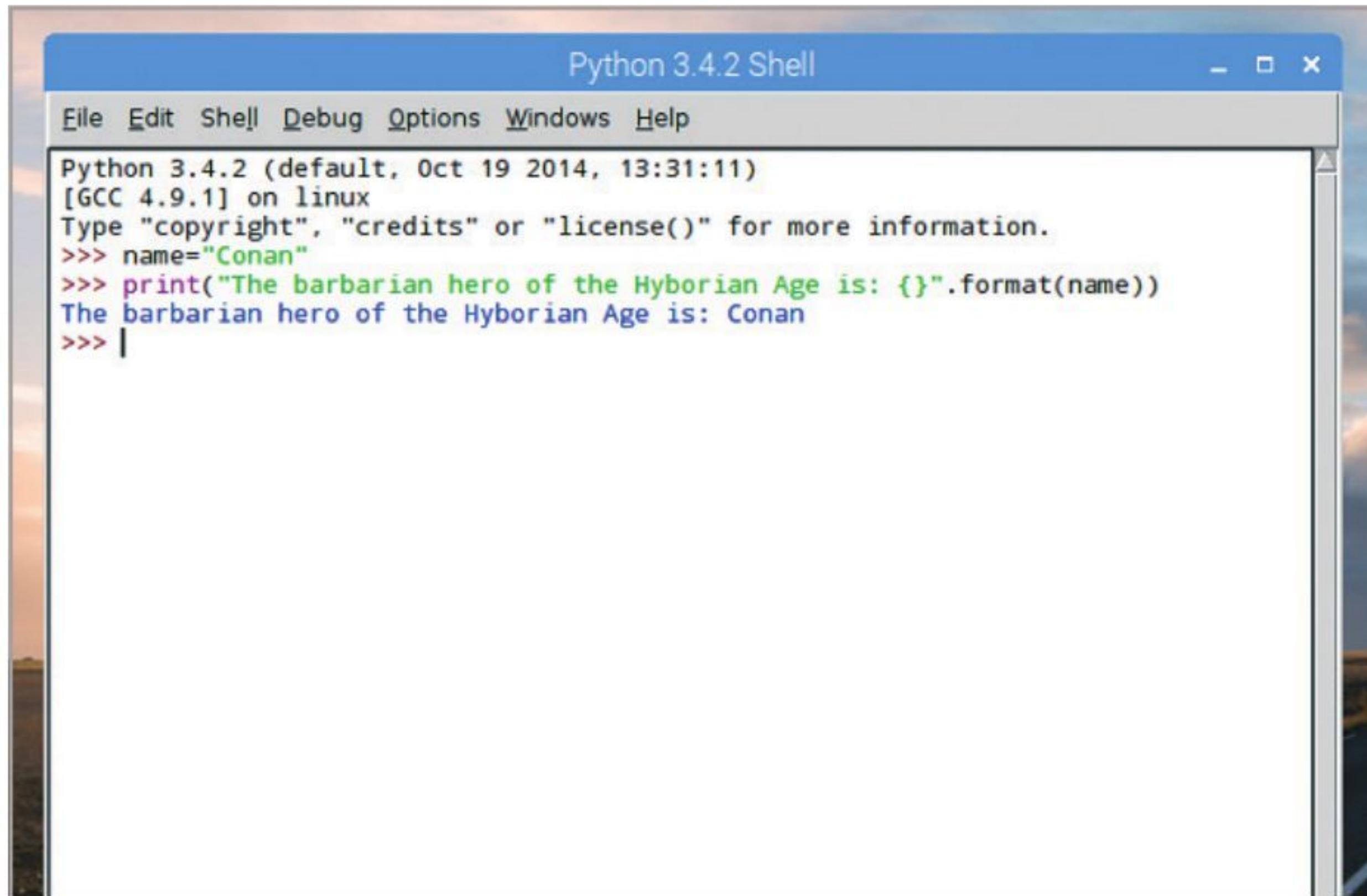
STRING FORMATTING

Since Python 3, string formatting has become a much neater process, using the `.format` function combined with curly brackets. It's a more logical and better formed approach than previous versions.

STEP 1

The basic formatting in Python is to call each variable into the string using the curly brackets:

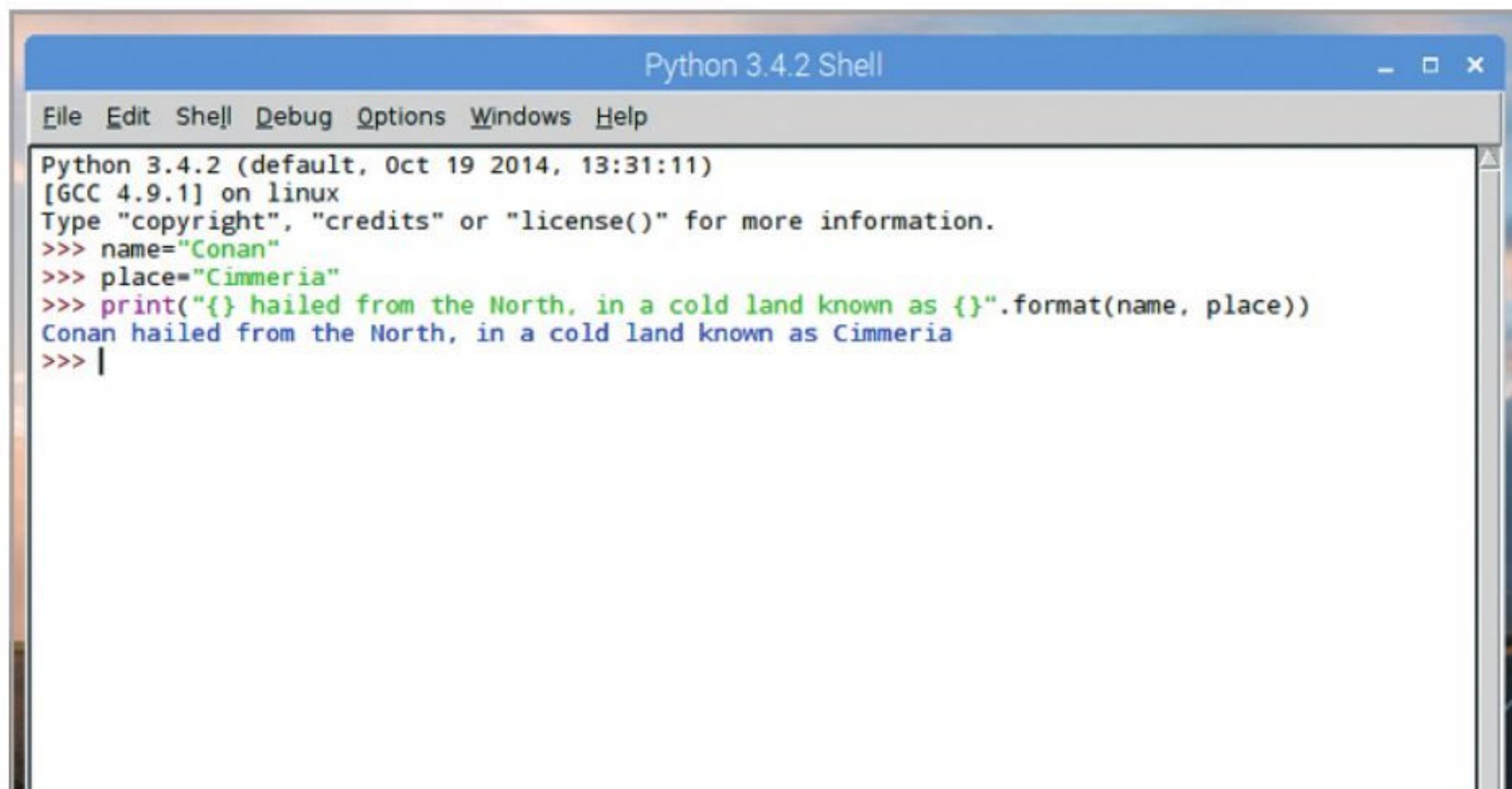
```
name="Conan"
print("The barbarian hero of the Hyborian Age is:
{}".format(name))
```



STEP 2

STEP 2 Remember to close the print function with two sets of brackets, as you've encased the variable in one, and the print function in another. You can include multiple cases of string formatting in a single print function:

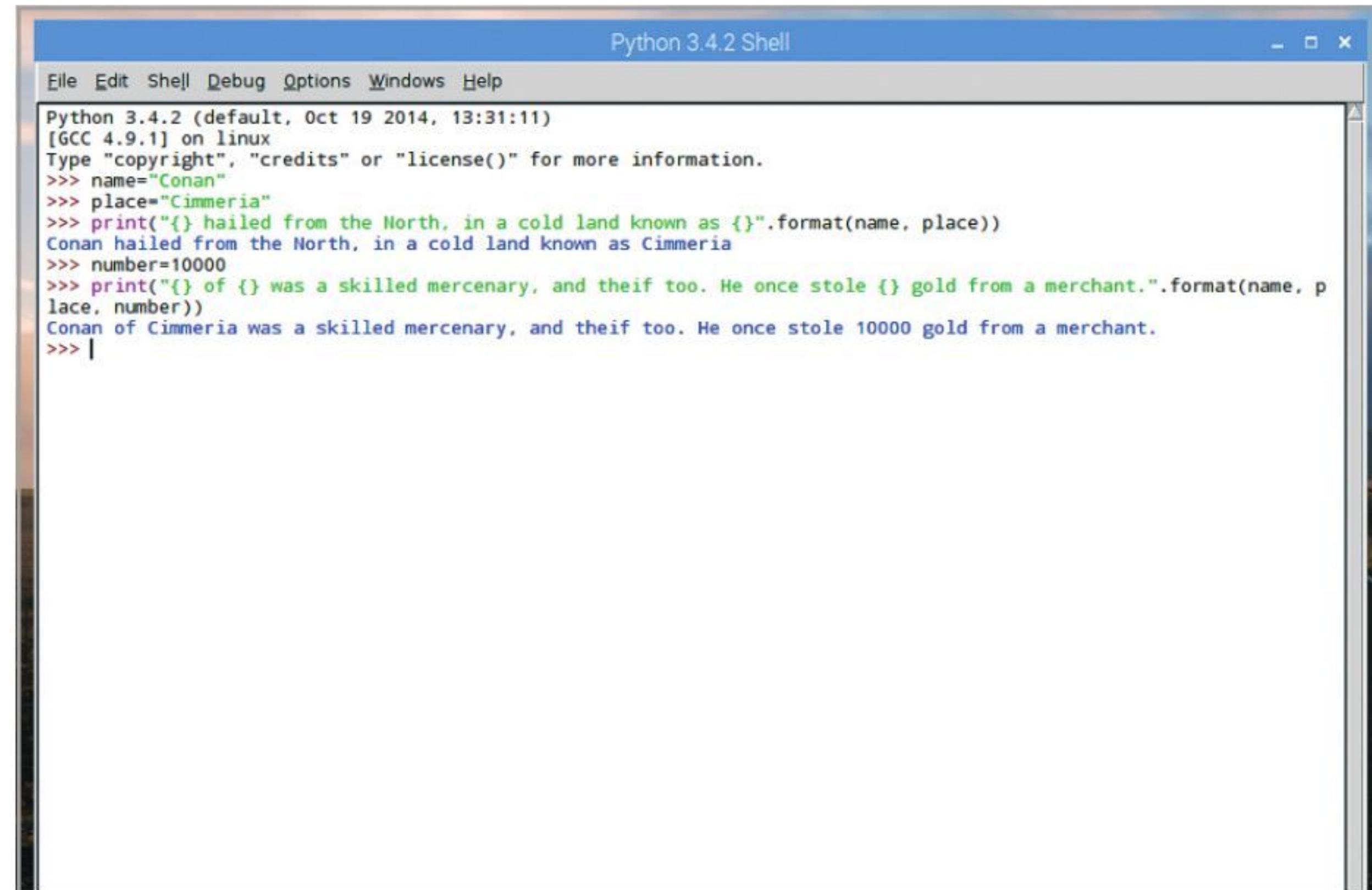
```
name="Conan"
place="Cimmeria"
print("{} hailed from the North, in a cold land
known as {}".format(name, place))
```



STEP 3

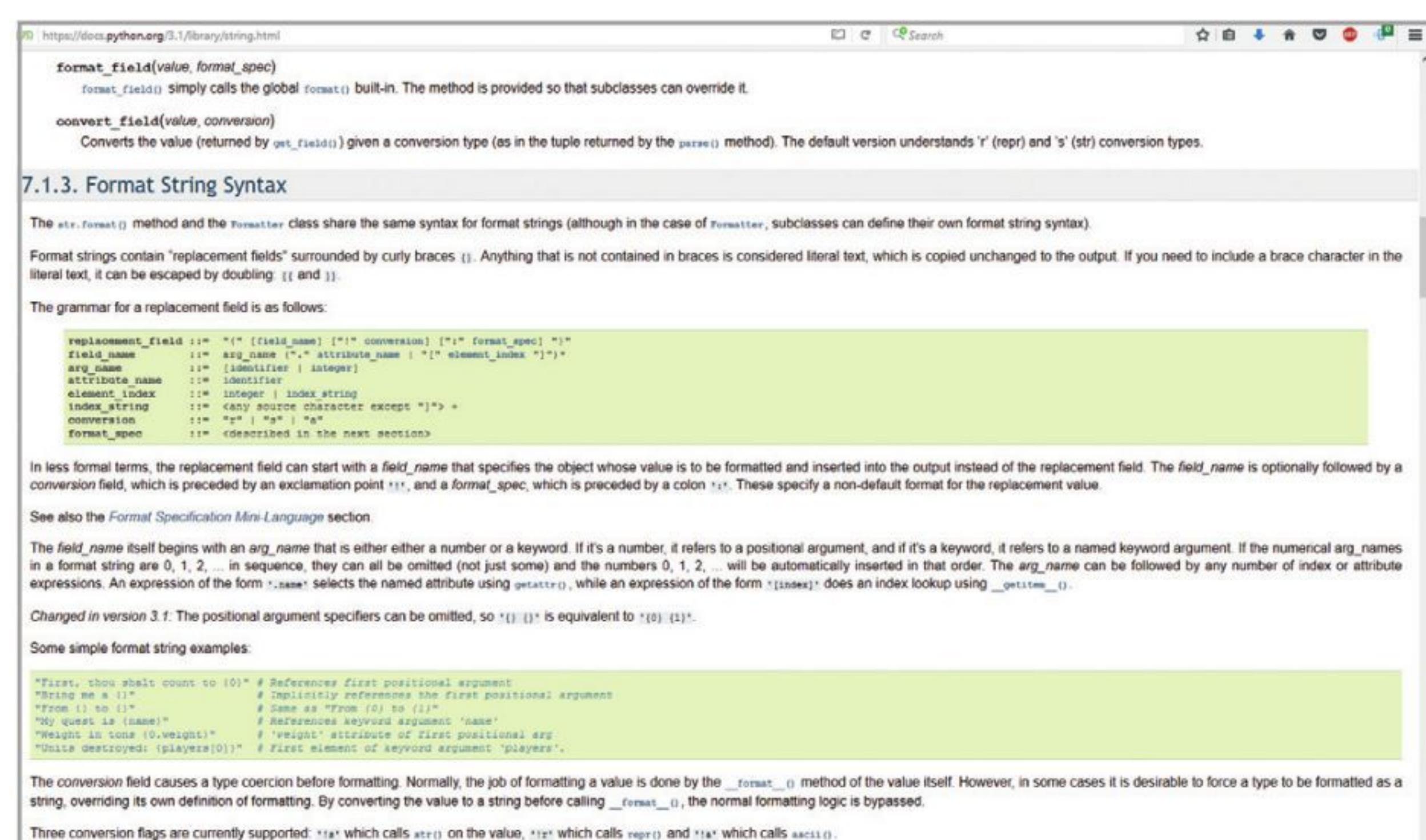
You can of course also include integers into the mix:

```
number=10000
print("{} of {} was a skilled mercenary,
and thief too. He once stole {} gold from a
merchant.".format(name, place, number))
```



STEP 4

STEP 4 There are many different ways to apply string formatting, some are quite simple, as we've shown you here; others can be significantly more complex. It all depends on what you want from your program. A good place to reference frequently regarding string formatting is the Python Docs webpage, found at www.docs.python.org/3.1/library/string.html. Here, you will find tons of help.



STEP 5

Interestingly you can reference a list using the string formatting function. You need to place an asterisk in front of the list name:

```
numbers=1, 3, 45, 567546, 3425346345
print("Some numbers: {}, {}, {}, {}, {}, {}".
format(*numbers))
```

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> numbers=1, 3, 45, 567546, 3425346345
>>> print("Some numbers: {}, {}, {}, {}, {}, {}".
format(*numbers))
Some numbers: 1, 3, 45, 567546, 3425346345
>>>
```

STEP 6

With indexing in lists, the same applies to calling a list using string formatting. You can index each item according to its position (from 0 to however many are present):

```
numbers=1, 4, 7, 9
print("More numbers: {}, {}, {}, {}".
format(*numbers))
```

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> numbers=1, 4, 7, 9
>>> print("More numbers: {}, {}, {}, {}".
format(*numbers))
More numbers: 9, 1, 7, 4.
>>>
```

STEP 7

And as you probably suspect, you can mix strings and integers in a single list to be called in the .format function:

```
characters=["Conan", "Belit", "Valeria",
19, 27, 20]
print ("{} is {} years old. Whereas {} is {} years old.".format(*characters))
```

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> characters=["Conan", "Belit", "Valeria",
19, 27, 20]
>>> print ("{} is {} years old. Whereas {} is {} years old.".format(*characters))
Conan is 19 years old. Whereas Belit is 27 years old.
>>>
```

STEP 8

You can also print out the content of a user's input in the same fashion:

```
name=input("What's your name? ")
print("Hello {}.".format(name))
```

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> name=input("What's your name? ")
>>> print("Hello {}.".format(name))
Hello David.
>>>
```

STEP 9

You can extend this simple code example to display the first letter in a person's entered name:

```
name=input("What's your name? ")
print("Hello {}.".format(name))
lname=list(name)
print("The first letter of your name is a {}".
format(*lname))
```

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> name=input("What's your name? ")
>>> print("Hello {}.".format(name))
Hello David.
>>> lname=list(name)
>>> print("The first letter of your name is a {}".
format(*lname))
>>>
```

STEP 10

You can also call upon a pair of lists and reference them individually within the same print function. Looking back the code from Step 7, you can alter it with:

```
names=["Conan", "Belit", "Valeria"]
ages=[25, 21, 22]
```

Creating two lists. Now you can call each list, and individual items:

```
print("{}[0] is {}[0] years old. Whereas {}[1] is {}[1] years old.".format(names, ages))
```

```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> names=["Conan", "Belit", "Valeria"]
>>> ages=[25, 21, 22]
>>> print("{}[0] is {}[0] years old. Whereas {}[1] is {}[1] years old.".format(names, ages))
Conan is 25 years old. Whereas Belit is 21 years old.
>>>
```