Python CheatSheet (For Linux)
Beginning Python: From Novice to Professional
Chapter 5(Conditionals, Loops, and Some Other Statements)

Printing with Commas

print can be used to print an expression, which is either a string or automatically converted to one. But you can actually print more than one expression, as long as you separate them with commas and a space character is inserted between each argument:

```
>>> print 'Age:', 42
Age: 42
>>> 1, 2, 3
(1, 2, 3)
>>> print 1, 2, 3
1 2 3
>>> print (1, 2, 3)
(1, 2, 3)
>>> name = 'Gumby'
>>> salutation = 'Mr.'
>>> greeting = 'Hello,'
>>> print greeting, salutation, name
Hello, Mr. Gumby
```

If you add a comma at the end, your next print statement will continue printing on the same line.

Importing Something

Usually, when you import something from a module, you either use **:**

```
import somemodule
```

or

```
from somemodule import somefunction
```

or

```
from somemodule import somefunction, anotherfunction, yetanotherfunction
```

or

```
from somemodule import *
```

The fourth version should be used only when you are certain that you want to import everything from the given module.
One can add an as clause to the end and supply the name you want to use, either for the entire module:

```
>>> import math as foobar
>>> foobar.sqrt(4)
2.0
```

or for the given function:

```
>>> from math import sqrt as foobar
>>> foobar(4)
2.0
```

Sequence Unpacking

You can perform several different assignments simultaneously:

```
>>> x, y, z = 1, 2, 3
>>> print x, y, z
1 2 3
```

you can use it to switch the contents of two (or more) variables:

```
>>> x, y = y, x
>>> print x, y, z
2 1 3
```

Actually, what I'm doing here is called sequence unpacking (or iterable unpacking). I have a sequence (or an arbitrary iterable object) of values, and I unpack it into a sequence of variables. Python 3.0 has another unpacking feature: you can use the **star operator (*)**, just as in function argument lists (see Chapter 6). For example, a, b, rest* = [1, 2, 3, 4] will result in rest gathering whatever remains after assigning values to a and b .

Chained Assignments

Chained assignments are used as a shortcut when you want to bind several variables to the same value. This may seem a bit like the simultaneous assignments in the previous section, except that here you are dealing with only one value:

```
x = y = somefunction()
```

which is the same as

```
y = somefunction()
x = y
```

Note that the preceding statements may not be the same as

```
x = somefunction()
y = somefunction()
```

For more information, see the section about the identity operator (is), later in this chapter.

Augmented Assignments

Instead of writing x = x + 1, you can just put the expression operator (in this case +) before the assignment operator (=) and write x += 1. This is called an augmented assignment, and it works with all the standard operators, such as *, /, %, and so on:

```
>>>x = 2
>>>x+=1
>>>x*=3
>>>x
6
```

Blocks: The Joy of Indentation

A block is a group of statements that can be executed if a condition is true (conditional statements), or executed several times (loops), and so on. A block is created by indenting a part of your code; that is, putting spaces in front of it.

The following values are considered by the interpreter to mean false when evaluated as a Boolean expression (for example, as the condition of an if statement):

```
False   None   0        ""       ()       []       {}
```

In other words, the standard values False and None, **numeric zero of all types (including float, long, and so on)**, **empty sequences (such as empty strings, tuples, and lists)**, and **empty dictionaries are all considered to be false**. Everything else is interpreted as true, including the special value True.

True and False aren't that different—they're just glorified versions of 0 and 1 that look different but act the same:

```
>>> True
True
>>> False
False
>>> True == 1
True
>>> False == 0
True
>>> True + False + 42
43
```

The Boolean values True and False belong to the type bool, which can be used (just like, for example, list, str, and tuple) to convert other values:

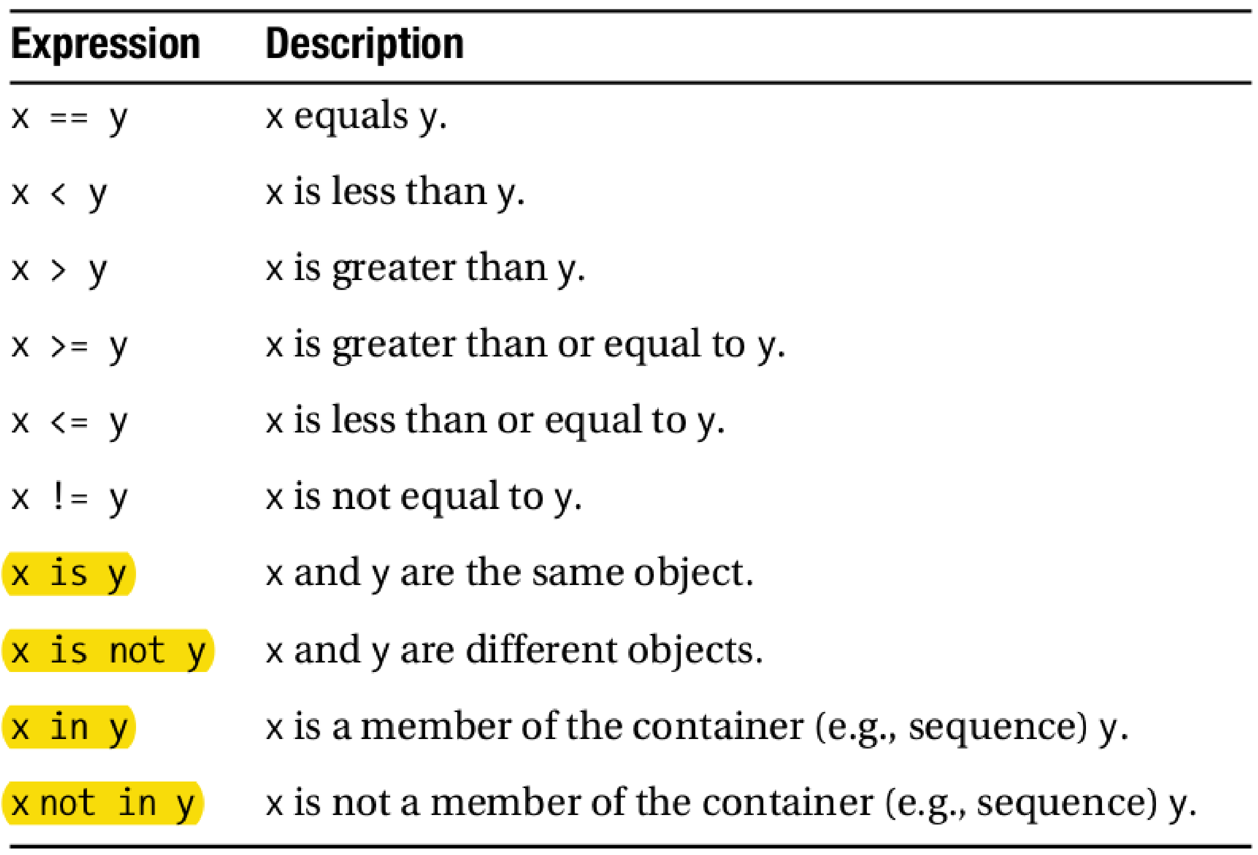
```
>>> bool('I think, therefore I am')
```

```
True
>>> bool(42)
True
```

Conditional Statement(if, else, elif{which as else if}, and nested if else), below example:

```
name = raw_input('What is your name? ')
if name.endswith('Gumby'):
        if name.startswith('Mr.'):
                print 'Hello, Mr. Gumby'
        elif name.startswith('Mrs.'):
                print 'Hello, Mrs. Gumby'
        else:
                print 'Hello, Gumby'
else:
        print 'Hello, stranger'
```

Table 5-1. *The Python Comparison Operators*



Expression	Description
`x == y`	x equals y.
`x < y`	x is less than y.
`x > y`	x is greater than y.
`x >= y`	x is greater than or equal to y.
`x <= y`	x is less than or equal to y.
`x != y`	x is not equal to y.
`x is y`	x and y are the same object.
`x is not y`	x and y are different objects.
`x in y`	x is a member of the container (e.g., sequence) y.
`x not in y`	x is not a member of the container (e.g., sequence) y.

If you stumble across the expression **x <> y** somewhere, this means **x != y** . The <> operator is **deprecated**, however, and you should avoid using it.

The Equality Operator
If you want to know if two things are equal, use the equality operator, written as a double equality sign, ==:

```
>>> "foo" == "foo"
True
>>> "foo" == "bar"
```

```
False
```

The single equality sign is the assignment operator, which is used to change things, which is not what you want to do when you compare things.

is: The Identity Operator

The is operator is interesting. It seems to work just like ==, but it doesn't:

```
>>> x = y = [1, 2, 3]
>>> z = [1, 2, 3]
>>> x == y
True
>>> x == z
True
>>> x is y
True
>>> x is z
False
```

Until the last example, this looks fine, but then you get that strange result: x is not z, even though they are equal. Why? Because is tests for identity, rather than equality. The variables x and y have been bound to the same list, while z is simply bound to another list that happens to contain the same values in the same order. They may be equal, but they aren't the same object.

To summarize: use == to see if two objects are equal, and use is to see if they are identical (the same object).

String and Sequence Comparisons

Strings are compared according to their order when sorted alphabetically:

```
>>> "alpha" < "beta"
True
```

Boolean Operators

if ((cash > price) **or** customer_has_good_credit) **and** not out_of_stock:
give_goods()

Assertions

There is a useful relative of the if statement, which works more or less like this (pseudocode):

```
if not condition:
        crash program
```

```
>>> age = 10
>>> assert 0 < age < 100
>>> age = -1
>>> assert 0 < age < 100
Traceback (most recent call last):
File "<stdin>", line 1, in ?
AssertionError
```

Loops:

while loop

example:

```
x = 1
while x <= 100:
        print x
        x += 1
```

for loop

The while statement is very flexible. It can be used to repeat a block of code while any condition is true. While this may be very nice in general, sometimes you may want something tailored to your specific needs. One such need is to perform a block of code for each element of a set (or, actually,

sequence or other iterable object) of values.
Example:

```
words = ['this', 'is', 'an', 'ex', 'parrot']
for word in words:
        print word
```

Python has a built-in function to make ranges for you:

```
>>> range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Ranges work like slices. They include the first limit (in this case 0), but not the last (in this case 10). Quite often, you want the ranges to start at 0, and this is actually assumed if you supply only one limit.
The **xrange** function works just like range in loops, but where range creates the whole sequence at once, xrange creates only one number at a time. 6 This can be useful when iterating over huge sequences more efficiently, but in general, you don't need to worry about it.

Iterating Over Dictionaries

To loop over the keys of a dictionary, you can use a plain for statement, just as you can with sequences:

```
d = {'x': 1, 'y': 2, 'z': 3}
for key in d:
        print key, 'corresponds to', d[key]
```

One great thing about for loops is that you can use sequence unpacking in them:

```
for key, value in d.items():
        print key, 'corresponds to', value
```

Parallel Iteration

Sometimes you want to iterate over two sequences at the same time. Let's say that you have the following two lists:

```
names = ['anne', 'beth', 'george', 'damon']
ages = [12, 45, 32, 102]
```

If you want to print out names with corresponding ages, you could do the following:
for i in range(len(names)):

```
print names[i], 'is', ages[i], 'years old'
```

Here, i serves as a standard variable name for loop indices (as these things are called). A useful tool for parallel iteration is the built-in function zip, which "zips" together the sequences, returning a list of tuples:

```
>>> zip(names, ages)
[('anne', 12), ('beth', 45), ('george', 32), ('damon', 102)]
```

Now I can unpack the tuples in my loop:

```
for name, age in zip(names, ages):
        print name, 'is', age, 'years old'
```

The zip function works with as many sequences as you want. It's important to note what zip does when the sequences are of different lengths: it stops when the shortest sequence is used up:

```
>>> zip(range(5), xrange(100000000))
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
```

Numbered Iteration

In some cases, you want to iterate over a sequence of objects and at the same time have access to the index of the current object. For example, you might want to replace every string that contains the substring 'xxx' in a list of strings. There would certainly be many ways of doing this, but let's say you want to do something along the following lines:

```
for string in strings:
        if 'xxx' in string:
                index = strings.index(string) # Search for the string in the list of strings
                strings[index] = '[censored]'
```

This would work, but it seems unnecessary to search for the given string before replacing it. Also, if you didn’t replace it, the search might give you the wrong index (that is, the index of some previous occurrence of the same word). A better version would be the following:

```
index = 0
for string in strings:
    if 'xxx' in string:
        strings[index] = '[censored]'
        index += 1
```

This also seems a bit awkward, although acceptable. Another solution is to use the built-in function

enumerate:

```
for index, string in enumerate(strings):
    if 'xxx' in string:
        strings[index] = '[censored]'
```

This function lets you iterate over index-value pairs, where the indices are supplied automatically.

Reversed and Sorted Iteration

They’re similar to the list methods reverse and sort (with sorted taking arguments similar to those taken by sort), but they work on any sequence or iterable object, and instead of modifying the object in place, they return reversed and sorted versions:

```
>>> sorted([4, 3, 6, 8, 3])
[3, 3, 4, 6, 8]
>>> sorted('Hello, world!')
[' ', '!', ',', 'H', 'd', 'e', 'l', 'l', 'l', 'o', 'o', 'r', 'w']
>>> list(reversed('Hello, world!'))
['!', 'd', 'l', 'r', 'o', 'w', ' ', ',', 'o', 'l', 'l', 'e', 'H']
>>> ''.join(reversed('Hello, world!'))
'!dlrow ,olleH'
```

Note that although sorted returns a list, reversed returns a more mysterious iterable object.

Break and Continue work as in C programing language and others.

List Comprehension—Slightly Loopy

```
>>> [x*x for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [x*x for x in range(10) if x % 3 == 0]
[0, 9, 36, 81]
```

You can also add more for parts:

```
>>> [(x, y) for x in range(3) for y in range(3)]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

■ Note Using normal parentheses instead of brackets will not give you a “tuple comprehension.” In Python 2.3 and earlier, you’ll simply get an error; in more recent versions, you’ll end up with a generator.

Nothing Happened!

Sometimes you need to do nothing. This may not be very often, but when it happens, it’s good to know that you have the pass statement:

```
>>> pass
>>>
```

Deleting with del

In general, Python deletes objects that you don’t use anymore (because you no longer refer to them through any variables or parts of your data structures):

```
>>> scoundrel = {'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}
>>> robin = scoundrel
>>> scoundrel
{'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}
```

```
>>> robin
{'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}
>>> scoundrel = None
>>> robin
{'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}
>>> robin = None
```

At first, robin and scoundrel are both bound to the same dictionary. So when I assign None to scoundrel, the dictionary is still available through robin. But when I assign None to robin as well, the dictionary suddenly floats around in the memory of the computer with no name attached to it. There is no way I can retrieve it or use it, so the Python interpreter (in its infinite wisdom) simply deletes it. (This is called **garbage collection**.)

Another way of doing this is to use the **del** statement.

This not only removes a reference to an object, but it also removes the name itself:

```
>>> x = 1
>>> del x
>>> x
Traceback (most recent call last):
File "<pyshell#255>", line 1, in ?
x
NameError: name 'x' is not defined
```

In fact, there is no way to delete values in Python—and you don't really need to, because the Python interpreter does it by itself whenever you don't use the value anymore.

Executing and Evaluating Strings with exec and eval

exec

The statement for executing a string is exec:

```
>>> exec "print 'Hello, world!'"
Hello, world!
```

However, using this simple form of the exec statement is rarely a good thing. In most cases, you want to supply it with a namespace—a place where it can put its variables. You want to do this so that the code doesn't corrupt your namespace (that is, change your variables). For example, let's say that the code uses the name sqrt:

```
>>> from math import sqrt
>>> exec "sqrt = 1"
>>> sqrt(4)
Traceback (most recent call last):
File "<pyshell#18>", line 1, in ?
sqrt(4)
TypeError: object is not callable: 1
```

Well, why would you do something like that in the first place? The exec statement is mainly useful when you build the code string on the fly. And if the string is built from parts that you get from other places, and possibly from the user, you can rarely be certain of exactly what it will contain. So to be safe, you give it a dictionary, which will work as a namespace for it.

You do this by adding in <scope>, where <scope> is some dictionary that will function as the namespace for your code string:

```
>>> from math import sqrt
>>> scope = {}
>>> exec 'sqrt = 1' in scope
>>> sqrt(4)
2.0
>>> scope['sqrt']
1
```

Note that if you try to print out scope, you see that it contains a lot of stuff because the dictionary called __builtins__ is automatically added and contains all built-in functions and values:

```
>>> len(scope)
2
>>> scope.keys()
['sqrt', '__builtins__']
```

In Python 3.0, exec is a function, not a statement.

eval

A built-in function that is similar to exec is eval (for “evaluate”). Just as exec executes a series of Python statements, eval evaluates a Python expression (written in a string) and returns the resulting value. (exec doesn’t return anything because it is a statement itself.) For example, you can use the following to make a Python calculator:

```
>>> eval(raw_input("Enter an arithmetic expression: "))
Enter an arithmetic expression: 6 + 18 * 2
42
```

You can supply a namespace with eval, just as with exec, although expressions rarely
rebind variables in the way statements usually do.