

Introduction to Python

Contributed by Kovacs Peter Tamas in Programming

Python is a very handy tool whenever you need to put together a small script that manipulates some files in a few minutes. Moreover, it is also useful for bigger projects, as you get all the power you need from data structures, modularization, object orientation, unit testing, profiling, and the huge API.

Python has connection with almost everything. You have very advanced string and regular expression handling, you have threading, networking (with many protocols built-in), compression, cryptography, you can build GUIs with Tcl/Tk, and these are just a few of the built-in features. If you look around on the Internet, you'll be surprised how many applications and libraries have Python bindings: MySQL, ImageMagick, SVN, Qt, libXML, and so on. There are applications that provide plug-in interface through Python, like Blender and GIMP. In case this is not enough for you, you can write extension modules for Python in C or C++ using the Python/C API, or in the reverse case, you can use the Python interpreter as a module of your native application, that is, you can embed Python into your software.

Python is widely used by serious people for serious purposes. I just list some of the more well-known names here: NASA, Industrial Light and Magic, Philips, Honeywell, TTTech. You can find several other success stories on Python's website.

Python is available for any platform you can imagine: Linux, Solaris, Windows, Mac, AIX, BeOS, OS/2, DOS, QNX, or PlayStation, for example. This is very good, since once you have a running code in Python, since once you need the same on another platform, you don't even have to touch it, and it just runs. And believe me, this is a very important aspect whenever you use it in your work. You just never know when you'll be asked whether your code runs on the X platform too.

If I succeeded in convincing you that learning Python is a good investment, go on, I will be happy to introduce it for you in a fast and straightforward way.

In the following, I assume that you already know at least one programming language, so you know what a variable and a loop is. That is, it is intended for people who would like to learn Python as a second, third, etc. programming language.

For testing the examples, I will use Python version 2.5. But in case you have a somewhat older version, don't worry, the examples are going to run on older versions too (2.3, 2.4), or if not, I will remind you beforehand.

Setting up your programming environment

To start Python programming, all you need is a text editor (if it has syntax highlighting for Python, that's even better), and the Python interpreter, which is usually installed with today's Linux distributions. If not, you just grab one from www.python.org or look for it in your package manager.

Of course, you can have more than a simple text editor. We all know that a handy Integrated Development Environment (IDE) can make life a lot easier. You can use PyDEV for Eclipse, Emacs, Komodo (a full Python IDE from ActiveState), or Vim, just to name a few.

Getting started

As I don't want to take your time with theory, syntax or anything like that, let's get started with

some practice right away. Some people say that once you can code hello world in a programming language, you are very close to be able to do anything you want. So here it goes:

```
#!/bin/python
print "Hello, world!"
```

Just enter these lines to a file, give the file a chmod +x, and run!

The first line specifies the interpreter to use to run the following code, as usual. In the second line, we send our greetings, and since nothing else follows, we just quit the program afterwards.

Staying for a second with print, I show you a very useful little feature, multi-line printing, which can be used for printing file headers, how-to-use texts, or any kind of multi-line blocks. You start the block with triple quotes, and finish it the same way.

It looks like this:

```
#!/bin/python
print """
First line.
Second line.
Third line.
"""
```

Variables

Enough of simple printing, there is no programming language without variables! So let's just introduce our first variables:

```
#!/bin/python
message = "Python is good for you!"
number = 24
print message
print number
```

As you can see, there is no need to tell Python the type of the variables. It will choose the appropriate type for you under the hoods. You rarely have to care about types in Python.

I shortly mentioned before that Python is very suitable for solving text-manipulation tasks. To show you how easy it is, I give some examples of tearing a text apart:

```
print message[0] # outputs 'P'
print message[10:14] # outputs 'good'
print message[19:] # outputs 'you!'
```

You can have a single character, a closed, or a half-open interval of characters in a string. From this example, you can also see that # is used for commenting. Staying at text manipulation, have a look at these:

```
print message.replace('you', 'your health')
print message.find('good')
```

The replace method replaces the first string found in message with the second one, then the return value gets printed. In the second line, find returns the position of the given substring in message. So the output will be:

Python is good for your health!

As you could see, our string variable has some methods too! This is because everything in Python is an object, having all the methods you'll ever need.

Just like with strings, you can also manipulate numerical values in a simple way:

```
print 3 * 2 + 5.5 # outputs 11.5
```

When doing thing like these, you don't have to worry about types, again. Everything is computed as you might expect from a well behaving programming language.

There is also a special case which allows you to multiply an integer with a string. So you can do crazy (but sometimes very useful!) things like this:

```
print 3 * "Hooray" # outputs "HoorayHoorayHooray"
```

Data structures

Now turn our attention to data structures. In my opinion, the time you have to take coding a specific task highly depends on the data structures you have at hand. The more convenient they are, the more you can concentrate on solving your real task, instead of worrying about the storage and layout of your data. Python has some very handy data structures I will show you.

Lists

Let's start with lists! A list is just a sequence of things. They don't have to be of the same type, you can just throw in anything you like. You can create a list by listing the items between [and], and separate the items with commas .

```
list = [3, 6, 'dog', 'cat']
print list[2] # outputs 'dog'
```

Do you remember what we did to strings? We can also do the same with lists:

```
print list[1:3] # outputs [6, 'dog']
print list[:3] # outputs [3, 6, 'dog']
```

We have some operations we can use on lists. Query length, append, insert, remove and search elements, count specific items, sort or reverse the list. Let me give you some examples!

The length of a list can be determined using the len() function. It also works for string the same way.

```
print len(list) # outputs '4'
```

Append is used for putting a new element to the end of the list, increasing the number of elements by one.

```
list.append('bear') # list now contains [3, 6, 'dog', 'cat', 'bear']
print len(list) # outputs '5'
```

We have a method for inserting elements to a list. In the first parameter, we pass the index of the

element before which we want to insert the element, which is supplied in the second parameter.

```
list.insert(2, 'cat') # list now contains [3, 6, 'cat', 'dog', 'cat', 'bear']
```

We can remove elements found by their value using the remove method. This is why the first line removes element 3.

We can also remove an element from the list found by its position. It can be done using the del statement, demonstrated in the second line.

```
list.remove(3) # remove 3
del list[0] # remove first element (6)
print list # now that we removed all the numbers, we have animals in the list only: ['cat', 'dog', 'cat', 'bear']
```

We can count the number of occurrences of an element using the count method. To find the index of a specific element in the list, you can use the index method.

```
print list.count('cat') # outputs '2', as we have two cats
print list.index('bear') # outputs '3', as poor bear is the last in the list
```

The whole list can be reversed in place using the reverse method (notice that the list actually changes here, we don't have to assign the return value to the list variable).

We can sort the elements of the list in the same manner, using the sort method.

```
list.reverse() # revenge!
print list # outputs ['bear', 'cat', 'dog', 'cat']
list.sort()
print list # outputs ['bear', 'cat', 'cat', 'dog']
```

We can also use the arithmetic operators to do simple manipulations on the list. For example, appending two lists can be done like this:

```
print [1, 2, 3] + [4, 5]
```

Since you can put anything in a list, another list is not an exception either. We append another list to our list:

```
list.append(['nested', 'list'])
```

Which will then become

```
['bear', 'cat', 'cat', 'dog', ['nested', 'list']]
```

Following this idea, you can imagine that making a two dimensional list (list of lists) is not a big deal.

Tuples

Now that we know enough about lists, let's introduce a somewhat strange structure: tuples. They can be used to store N pieces of information, packed together. You can pack several values in a tuple, then unpack them later. For example, it's for storing the fields in a record, or coordinate pairs. You list the elements of a tuple between (and), and assign it to a variable. Then, when you need the values inside the tuple, you unpack it the reverse way, list the variables that should receive the individual values between (and), but you do this on the left hand side of the assignment! Looks

pretty strange, isn't it?

```
person = ('Jack', 'Nicholson', '1937') # pack
print person
(firstname, familyname, year) = person # unpack
print firstname # outputs 'Jack'
```

```
x = 2.3
y = 6.5
point = (x, y)
```

Dictionaries

Dictionaries are for storing values that can later be accessed directly using a key. This key can be either numerical or string, and the keys can be chosen arbitrarily. That is, you don't have to use 0-based, incrementing indices to point to the elements, like in an array.

It's similar to `std::map`, to those who know C++ and the Standard Template Library.

To use a dictionary, we have to initialize it first. For this example, we will store ages of people in the dictionary (that is, we will do a `string` → `int` mapping).

We just list the key-value pairs between `{` and `}`, then we can query the elements of the dictionary using `[]`.

```
ages={'Peter':25, 'Zsuzsi':24}
print ages['Zsuzsi'] # outputs '24'
```

Of course, we can also start with an empty dictionary, and assign the values later, using the same syntax that we used for query.

```
ages={}
ages['Peter'] = 25
ages['Zsuzsi'] = 24
```

Please note that even if you want to start with an empty dictionary, you have to create one first (using `dictionary_name={}`)! That is, you cannot just assign a value to a non-existing dictionary, which would lead to an error.

Existing values can be modified in an intuitive way; you just reassign the new value to the already existing key.

```
ages['Peter'] = 26 # Peter gets one year older
print ages['Peter'] # outputs '26'
```

If you want to see the contents of a dictionary, you don't have to iterate over the elements, `print` handles this for you. It's simple like this:

```
print ages # outputs the following:
{'Zsuzsi': 24, 'Peter': 26}
```

It's also possible to delete elements, using the `del` statement.

```
del ages['Peter']
print ages # outputs '{'Zsuzsi': 24}'
```

You can make a list of the available keys, and values. The methods to do this are `keys()` and `values()`, respectively.

```
ages={'Peter':25, 'Zsuzsi':24}
print ages.keys() # outputs ['Zsuzsi', 'Peter']
print ages.values() # outputs [24, 25]
```

There are methods for clearing, copying, and querying whether a key exists already. They are called `clear()`, `copy()`, and `has_key()`.

Sets

I think I don't really have to explain what a set is, as everyone should know them from mathematics. It's simply a pile of elements that do not have ordering and do not contain duplicates.

A set has to be initialized with the elements of a list. Since you already know what a list is, we do this in one step. Just like with dictionaries, `print` can handle a set as it is.

Once we have a set, I show the first useful feature of sets: testing whether an element is in the set.

```
inventory_carpenter=set(['helmet', 'gloves', 'hammer'])
print inventory_carpenter # outputs set(['helmet', 'hammer', 'gloves'])

print 'gloves' in inventory_carpenter # outputs 'True'
```

Since sets are interesting only if we have more than one of them, let's introduce another one! Once we have that, we can immediately see what are the elements that both sets contain (intersection).

```
inventory_highscaler=set(['helmet', 'rope', 'harness', 'carabiner'])

print inventory_carpenter & inventory_highscaler # outputs 'set(['helmet'])'
```

Similarly, we can have the union of sets (using `|`), difference (using `-`), or symmetric difference (using `^`).

For sets, you don't really need anything else, as you can do every meaningful operation using the ones above. For example, to add a new element, you can use union.

```
inventory_carpenter = inventory_carpenter | set(['nails'])
```

Using the interpreter interactively

If you would like to try out the things you've learnt right now, you might appreciate that the interpreter can be used in an interactive way. In case you use it like that, you don't have to enter your commands to a file, then save and run it, just tell something to Python, and get the response immediately.

All you have to do is to invoke the interpreter by typing 'python' to your shell.

```
kovacsp@centaur:~$ python
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17)
[GCC 4.1.2 20060715 (prerelease) (Debian 4.1.1-9)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Once you are here, you can just type anything you like, it will get interpreted immediately. The

good new is that you don't even have to use print if you want to see the value of a variable, just type the name of it.

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

pySerial

Overview

This module encapsulates the access for the serial port. It provides backends for Python running on Windows, Linux, BSD (possibly any POSIX compliant system) and Jython. The module named "serial" automatically selects the appropriate backend.

It is released under a free software license, see LICENSE.txt for more details.

(C) 2001-2003 Chris Liechti cliechti@gmx.net

The project page on SourceForge and here is the CVS repository and the Download Page.
The homepage is on <http://pyserial.sf.net>

Features

- same class based interface on all supported platforms
- access to the port settings through Python 2.2 properties
- port numbering starts at zero, no need to know the port name in the user program
- port string (device name) can be specified if access through numbering is inappropriate
- support for different bytesizes, stopbits, parity and flow control with RTS/CTS and/or Xon/Xoff
- working with or without receive timeout
- file like API with "read" and "write" ("readline" etc. also supported)

The files in this package are 100% pure Python. They depend on non standard but common packages on Windows (win32all) and Jython (JavaComm). POSIX (Linux, BSD) uses only modules from the standard Python distribution)

The port is set up for binary transmission. No NULL byte stripping, CR-LF translation etc. (which are many times enabled for POSIX.) This makes this module universally useful.

Requirements

- Python 2.2 or newer
- win32all extensions on Windows
- "Java Communications" (JavaComm) extension for Java/Jython

Installation

Extract files from the archive, open a shell/console in that directory and let Distutils do the rest:
python setup.py install

The files get installed in the "Lib/site-packages" directory.


```

        #can specify a device string, note
        #that this isn't portable anymore
        #if no port is specified an unconfigured
        #an closed serial port object is created
baudrate=9600,      #baudrate
bytesize=EIGHTBITS, #number of databits
parity=PARITY_NONE, #enable parity checking
stopbits=STOPBITS_ONE, #number of stopbits
timeout=None,      #set a timeout value, None for waiting forever
xonxoff=0,        #enable software flow control
rtscts=0,         #enable RTS/CTS flow control
)

```

The port is immediately opened on object creation, if a port is given. It is not opened if port is None.

Options for read timeout:

```

timeout=None      #wait forever
timeout=0         #non-blocking mode (return immediately on read)
timeout=x         #set timeout to x seconds (float allowed)

```

Methods of Serial instances

```

open()           #open port
close()          #close port immediately
setBaudrate(baudrate) #change baudrate on an open port
inWaiting()     #return the number of chars in the receive buffer
read(size=1)    #read "size" characters
write(s)        #write the string s to the port
flushInput()    #flush input buffer, discarding all it's contents
flushOutput()   #flush output buffer, abort output
sendBreak()     #send break condition
setRTS(level=1) #set RTS line to specified logic level
setDTR(level=1) #set DTR line to specified logic level
getCTS()        #return the state of the CTS line
getDSR()        #return the state of the DSR line
getRI()         #return the state of the RI line
getCD()         #return the state of the CD line

```

Attributes of Serial instances

Read Only:

```

portstr         #device name
BAUDRATES      #list of valid baudrates
BYTESIZES      #list of valid byte sizes
PARITIES       #list of valid parities
STOPBITS       #list of valid stop bit widths

```

New values can be assigned to the following attributes, the port will be reconfigured, even if it's opened at that time:

```

port           #port name/number as set by the user
baudrate       #current baudrate setting
bytesize      #bytesize in bits
parity         #parity setting
stopbits       #stop bit width (1,2)
timeout        #timeout setting
xonxoff        #if Xon/Xoff flow control is enabled
rtscts         #if hardware flow control is enabled

```

Exceptions

serial.SerialException
Constants

parity:

- serial.PARITY_NONE
- serial.PARITY_EVEN
- serial.PARITY_ODD

stopbits:

- serial.STOPBITS_ONE
- serial.STOPBITS_TWO

bytesize:

- serial.FIVEBITS
- serial.SIXBITS
- serial.SEVENBITS
- serial.EIGHTBITS