Getting to know Python

crappy title by SolarBear

Contents :

Chapter 1
Introduction to programming

Note : this chapter is optional. If you want to program right away, just skip it and go to chapter 2.

Let's face it : programming has an uncanny aura. People are confused by computers, and just getting a printer to work can be a never-ending quest to some people, so the mere thought of programming these monstrous beasts would never even occur to them.

Still, somebody has to code all these excellent (?) programs you're using : from the operating system underlying your PC to the file viewer you're using to read this file. Programming is everywhere, and millons of users on the planet have at least a vague understanding of basic programming principles. Not everybody has the know-how to make complex pieces of software, though.

So the first question one must ask is : why the heck do we program ? Some do it out of curiosity, others as a hobby, some others do it as a living. But regardless of their status, people program for either or both of the following reasons :

- To solve a problem
- To provide a better solution to a solved problem

Think about it : either the program you're making exists or it does not (master of logic, aren't I ?). If it doesn't, you probably aren't going to code it for your own pleasure. And if it does, most people won't go as far as to code anew a satisfying solution to that same problem. Take email clients as an example. In the beginning, when the web was just born, mail clients appeared and provided the basic features people needed for mail manipulation : receiving and sending mail, address books, storage on the local machine, etc. But with time, these simplistic applications became out of date : HTML was integrated into emails, security became a more serious concern, multimedia content was added with technologies such as Shockwave or Flash, etc. So updates became necessary. A “better solution” to the mail problem, if you wish.

What is a programming language ?

Computers are the dumbest objects the human race as ever made : they understand NOTHING but 0's and 1's. However, there is ONE thing they're good at, and it's doing what they're told. But sending commands as a series of binary numbers isn't really most people's idea of “user-friendliness” ; just imagine punching billions of '0' and '1' a second on a 2-key keyboard. In the 1950's, when programs were really small and simplistic, this could be done using cards, but as systems became more and more complex, it became obvious we needed something more “useable”. Then someone came with the idea of a human-readable language that would be translated to binary-understandable files. Assembly was born, and all was good for some time.

However, systems grew exponentially, and ASM programs were difficult to maintain,

but also and especially to understand if they weren't correctly commented :

```
PUSH ESP
PUSH 1
PUSH 0
PUSH app1.0040200C
PUSH 80000002
CALL <JMP.&ADVAPI32.RegOpenKeyExA>
TEST EAX,EAX
JNZ app1.004011AF
LEA EAX,DWORD PTR SS:[ESP+8]
PUSH EAX
PUSH exepuzz1.00402084
LEA EDX,DWORD PTR SS:[ESP+C]
PUSH EDX
PUSH 0
PUSH exepuzz1.00402021
MOV ECX,DWORD PTR SS:[ESP+14]
PUSH ECX
```

Even experienced coders would need to sit down and think for a second about what that piece of code does. So while ASM programming definitely was a solid step in the right direction, we were still lacking readablity. Some developpers quickly understood that and began working on something more useable. And they came up with FORTRAN. (For an idea of how programming languages evolved, check the very complete timeline at http://www.levenez.com/lang/history.html#01). And from FORTRAN and a few other languages like Algol and Lisp, we ended up with the variety of languages we know today.

Why Python ?

A very legitimate question a lot of people keep asking me. Some well-known people, like Eric S. Raymond had to answer the exact same question, and you can find Mr. Raymond's answer at http://www.linuxjournal.com/article.php?sid=3882

I do have a few reasons of mine :

- Compiled languages (e.g. C/C++) are not cross-platform, except for the most trivial of cases. On the other hand, interpreted languages like Python, and others like Tcl, Ruby or Perl, are completely cross-platform since the interpreter deals with platform specifics. You just have to code and let Python handle most of the trouble.
- Also, interpreted languages allow easier debugging. Expert gdb and UNIX users would argue that it's easy, really, forgetting how much time they spent reading man pages to try to figure out how in the HELL was it was supposed to work. (Sorry, bit of rambling here.)
- Python is extremely readable. Somebody said : “Python is executable pseudocode” and he/she was completely right. While you may spend some time trying to decipher a complex C script dealing with pointers and memory allocation, Python takes care of everything for you, thus simplifying your job AND leaving code that's actually understandable to other programmers – although this does not mean you needn't comment your code !
- Python is heavily and well documented. Some languages with let you figure everything out by yourself, but Python's docs are insanely complete for the most part.

- Python is... fun. I can't really explain that one, sorry. Just try it out by yourself. You'll see what I mean.

How do I begin ?

Well the easy answer would be : install Python ! Go to www.python.org and grab the latest version of Python for your OS. Install is straightforward : Windows users will find their typical installation wizard, Linux users will get through installation via the usual ./configure, make and make install gig or get the latest package for your distribution. As for Mac users, just look into your system : OS X users will most probably have it installed by default, while users of OS 9 and previous should look for MacPython (Google your way to it) or compile it from the source if they're motivated and Mac-savvy enough.

When you're done installing, look at what you got : the interpreter shell, the documentation and the IDLE user interface. Take some time to discover these fascinating tools.

The interpreter shell

Except maybe for Lisp programmers, you may not be used to have an interactive interpreter. What's it use ? Well, you can use it to enter your scripts, even though that's not really useful, but it does allow some on-the-fly testing, if you want to test some short code snippets. Some people even use it as a calculator. So, how do you use it ? Easy.

```
>>> 2 + 2
4
>>> “My name is John.”
My name is John.
>>> 3 % 2
1
```

Maybe you're not fully understanding what's going on, but the point is : you can input Python instructions and the shell will execute them. You could even put a simple function and execute it :

```
>>> def foo(a):
...         print a
>>> b = “Hullo!”
>>> print b
Hullo!
```

Voilà ! You have some easy to use way of coding. The preferred way would be to put functions inside Python script files, but more on that later.

The documentation

Python's docs are available online on the official Python website ; however, you do want to save them bandwidth, don't you ? I know you do, so you'll want to use your own. Python's documentation is extremely complete : every module is fully documented and there's even a tutorial included, which is quite good but has some downsides (sorry Guido !), like the lack of

example code. It's still a very valuable reference. Windows' docs even come in compiled HTML format for easier navigation.

IDLE

Any serious, and even not-so-serious, programmer needs a tool for syntax highlighting, and IDLE comes packaged with Python. IDLE is a quite good editor for beginners as well as small- to medium- sized projects. Simply put : good enough for most of us. It's built using Tkinter, Python's “official” GUI programming kit, so you can have a good idea of what Python is able to do.

An application built in Python to generate Python code. How ironic.

Now, enough chit-chat : let's get to programming.

Chapter 2
The obligatory “Hello world!”

It has become tradition as years passed to make our very first program as an “Hello world !” program. What we're trying to achieve is to output these words on a console window. Some languages won't let you do it so easily. Look at the Java code:

```
import system;
public class HelloWorld
{
   public static void main()
   {
      System.out.println(“Hello world!”);
   }
}
```

Now witness Python's sheer simplicity !

```
#!/usr/bin/env python
print 'Hello world !'
```

Easy enough ? Indeed it is, but let's review these two lines, just to be sure.

```
#!/usr/bin/env python
```

This line is unnecessary for Windows users as well as other users who put their Python interpreter in their PATH global variable. It tells your shell what interpreter to use. If you didn't mention it, your system wouldn't even know what to do with the file, or would execute it as a shell script, giving various results depending on the shell you're using. It's just a comment to most systems anyway. Your path may be different, though.

```
print 'Hello world !'
```

This is pretty self-explanatory. The print built-in function simply puts its arguments to the screen, and ends it with a newline.

So what to do with this code ? Simply copy-paste it to IDLE, save it as a .py file somewhere on your PC and execute it (Linux users : be sure to chmod +x your file !). What happens ?

- Windows, not on console : a console windows pops up and then disappears. Wait, did anything go wrong ? No, calm down, this is a common problem. When your Python file is launched, it opens a console window, executes itself and then closes when it's done. So it outputs “Hello world!” and then closes. Just put raw_input() on a new line at the end of your file. This way, you'll need to press enter before the window closes. A very useful trick, even for more complex programs. We'll get back to functions later.
- Any OS on console : “Hello world!” is shown, and your return to command line.

Chapter 3
Variables and data types

Didja fail your high school algebra classes ? Whether or not you did is unimportant : we're talking about basic definitions of variables. Nothing about calculus or vectors here. Unless you politely ask for it, of course.

What is a variable ?

Again, I got into the bare basics but to understand complex concepts, the simple ones must be thoroughly understood. So, what's a variable exactly ?

In the mathematical sense of the word, a variable is simply an undertermined term of which all possible values are determined. As you can imagine, this isn't of much use to us. However, for a computer, a variable is a binary value placed somewhere in memory. See it as a container. This variable can have different types. You probably are already familiar with most of these types ; the others are really simple. Let's cover the essentials.

Integers

Integers are numbers like 0, 1, 2, -3 and 1,004,354. Any number without a decimal is an integer. They can be positive or negative. Here's some basic arithmetic using integers. Note : the multiplication operator is the star, *, and the division operator is the slash, /.

```
>>> 2 + 3
5
>>> -2 - (-3)
1
>>> -2 * 4
-8
>>> 3 / 4
0
```

What ? Three divided by four gives 0 ? Isn't that strange ? Not at all. When you divide a number by another, Python takes care of type conversion, if need be. Here, we divide an integer by another integer ; clearly, we'll get an integer as a solution. So when you divide an integer by another, you'll receive only the integer part of the answer. You can complement this operation using the modulus operator, %, which returns the remainder of the division. Here are some more example, just to be sure you're getting what I mean.

```
>>> 5 / 3
1
>>> 7 / 8
0
>>> 5 % 4
1
>>> 9 % 3
0
```

Exercises

- Try to guess the output of 7 / 4 + 7 % 4. After you think you're right, enter it in the Python shell and see if you were right or wrong.
- Again, try to guess the output of 4 / 3 * 3. Check your answer.

Long integers

Longs, as they're called for short, are really nothing more than longer integers (duh), meaning that while they take more memory, they also allow for greater numbers than our basic integers. Older version of Python will return an error when an integer becomes too big, but more recent version (2.0 and over, I guess) will convert it automatically to long.

Most day-to-day applications shouldn't need longs, but if, for some reason, you just need to declare a long integer, do it by putting 'l' or 'L' at the end.

```
>>> a = 0L
>>> print a
0
```

As you can see, Python is wise enough not to print the L.

Floating point numbers

Most commonly called “floats”, these numbers possess at least one digit after the usual decimal dot. Note to European readers : you may be used to noting numbers using a comma, such as 1,5 ; Python uses american dot notation, 1.5 , just like most programming languages. Operators are the same as for integers ; anyway, here a few more examples put through the shell :

```
>>> 4.0 / 1.5
2.6666666666666665
>>> 3 / 1.5
2.0
>>> 4 / 3.0 * 3
4.0
>>> 5.0 % 1.5
0.5
```

Math purists will argue that the modulus operator isn't defined for non-integers, and they would be right. But do I care ? Not at all. Nor do Python creators, I guess. Thus I pull my tongue at you. Bleh.

You can also use floats using scientific notation, if you so wish. This especially useful in scientific applications or to handle very large numbers : TODO

Exercises

TODO

Complex numbers

Just a quick note for the interested people : Python has a built-in type for complex numbers. It isn't of much use outside complex analysis and engineering, but it could still be useful to some of you. If you don't know what's a complex number, no need to read further : I won't go into the theory. Note that Python uses j as the square root of -1, while some math books would use i. Here are some examples. Put complex expressions between parentheses.

```
>>> (2 + 1j) - (4 - 6j)
(-2+7j)
>>> (1-3j) * (-3-4j)
(-15+5j)
```

If you're interested on this subject, consult the Python documentation.

Strings

The complete, exact name would be “character strings” but programmers are lazy people, and since they're not using strings of anything else... well... you get the idea. Some programming languages separate characters and strings as two different types, but Python does not. Strings are created by puttings characters between simple or double quotes : it makes no difference at all. So 'a' is the same as “a”.

Obviously, you can't multiply strings or use modulus on them. However, some mathematical operators are used with strings : + for concatenation, * for repetition.

```
>>> 'a'
a
>>> “a”
a
>>> 'a' + “b”
ab
>>> 'a' * 3
aaa
>>> “b” * 7
bbbbbbb
```

Be warned though ! If you begin your string with a single quote, you can't put any inside of your string, except by placing a backslash (\ ) before it. The backslash is called the insertion character ; more on that later.

Boolean values

There are only two of those : True and False. We'll discuss boolean values and operators in the next chapter.

Assigning values to variables

Getting tired of reading about types ? You should be. Now let's get to variables. How do you create one ? Creating a variable requires a name for that variable as well as a value for it. Simple heh ? And it's simple to do. Give your variables a meaningful name ! Don't call all of them a or i_3 : long programs quickly get unmanageable otherwise. Let's use a real world case, using the best human interaction algorithms available. (Yeah right.)

We want to make a script that will ask our name and age and then repeat it back to us. So we'll need to variables : one name that will contain our name as a string, and an age one that will store our... age. Ok I'll just shut up and put the code here.

```
print 'What is your name, sir ?'
name = raw_input()
print 'And what is your age ?'
age = input()
print 'So your name is ', name, ' and you are ', age' year(s) old.'
```

What did we learn here ?

- Variables are given a value via the affectation operator =. The variable's name is put on the left and the value put in it must be placed on the right. That value could be a literal value (e.g. a = 3), another variable (e.g. a = 3, b = a so b = 3) or a function's result, such as in the case above.
- The raw_input() function is very important for getting user input. It returns what the user entered as a string, while the input() function returns that input but in the correct type (integer, float, etc.)
- The print function can take more than one argument. Separated by commas, Python prints them one after another, putting a newline only at the end of the sequence.

So, what are valid variable names ? It must begin by an alphabetical character or an underscore, and other characters can be any alphanumerical characters or the underscore. Underscores are generally used to separate words, like in amount_of_meat.

Typecasting

It just happens sometimes : you need to use an integer but you only have a float available. What would you do then ? I asked what WOULD you do, punk ?

You use typecasting, that's the correct answer. Typecasting, as you may have already understood, is the act of transforming data into data of a different type. Here are some self-explanatory examples :

```
>>> int(4.0)
4
>>> float(4)
4.0
>>> string(4)
4
>>> int('4')
4
```

You can use type conversion for any of Python's built-in types. Just the type's name as a function and put what you're trying to convert as an argument. More on functions later.

Lists and tuples

These two types are very alike, and yet are pretty much different.

TODO

Chapter 4
Conditional and looping statements

As you may imagine, there's more to Python than variables and their types. Our “programs” are static : we need them to take decisions ! To progress further, we'll need an actual, real-world scenario for program building. So here's the job we've been offered :

Excellent, now we have something to work on. But how are we gonna do that, with the limited knowledge we have now ? Impossible, I tell ya. So we need to introduce one more concept.

Conditional statements

Yep, conditionals. It just so happens that, in some situations, a program must act in a way but must behave in another way in another. Programming languages evaluate conditions to see what to do : in our case, we need to check if the customer is 18 or over. In that case, he can enter ; otherwise, he receives a polite but firm message telling him to get lost.

As usual, here's the code, followed by an explanation :

```
print 'Welcome to the Zanzi Bar !'
age = input('Please enter your age : ')
if age >= 18:
  print 'You may enter. Have fun !'
else:
  print “Go see yer mamma, lil' baby.”
```

Nothing difficult here, I guess you can figure it out by yourself, but for the sake of completeness...

- As you may have noticed, you can pass a string to input() or raw_input(). That string will appear right before the user's entry.
- What's that '>=' thingy ? It's the “greater than or equal” operator, one of many other

operators we'll study in the next section. It checks if the l-value (value on the left of the operator) is greater than the r-value (value on the right) OR if they're equal. If that's the case, it returns True ; otherwise, it returns False. Those are the boolean values we mentionned in the previous chapter.

- So, what happens ? if the age the user inputed is greater than or equal to 18, the program outputs a message allowing them to enter (and hopefully, would enter the door) ; else, they're given the boot in the meanest of ways.

Easy huh ? if, else. There's also the elif keyword, that will accept another condition if that of the original if wasn't True. Here's the same code but with elif :

```
# Cyber Bouncer 1.0
print 'Welcome to the Zanzi Bar !'
age = input('Please enter your age : ')
if age == 18:
  print 'Just turned 18, heh ? Come in!'
elif age > 18:
  print 'You may enter. Have fun !'
else:
  print “Go see yer mamma, lil' baby.”
```

First, an important note: the “=” operator is NOT the same as “==”.

- “=” is the affectation operator, that assigns the r-value to the l-value. It returns nothing.
- “==” is the comparison operator. If two variables or objects are the same, it returns True and returns False otherwise.

So what will happen here is pretty much the same as before, except we split the comparison in two. if the customer is 18, he receives a welcoming message. If, and only if, that condition wasn't met (i.e. customer's age is different from 18), then we skip to the next conditional statement. If you didn't figure it out by yourself yet, elif is short for “else if”. So, else if the customer's age is greater than 18, he'll receive some message, and else receives a “Shoo!” message.

Note to C/C++/Java programmers:
- Python does not support C's **switch() case :... case :.... default : ...** . You'll need to use **if ... elif ... elif ... else** instead.

Logical operators

We've seen a few of them, but you must know all of them like the palm of your hand. Most of 'em, anyway.

Equals	==
Not equal to	**!=**
Greater than	**>**
Greater or equal to	**>=**

Equals	==
Lesser than	<
Lesser or equal to	<=
Invert	~
Logical NOT	**not**
Logical AND	**and**
Logical OR	**or**
Logical XOR	^

I won't go and describe the mathematical operators ; I believe the reader is intelligent enough to figure this out by himself if he passed grade 4 math. However, the last few operators must be studied since they're not known to all – or at all.

~

This operator is seldom used but could of use to some readers using low-level bit operations. What it does is simply invert the bits of any value ; simply put, it takes a binary value, such as 001101010 and replaces ones by zeros and vice versa. So our number would become 110010101.

```
>>> ~1
-2
>>>~-3
2
```

NOT

The NOT unary logical operator takes a boolean value and inverts it.

```
>>> not True
False
>>> not 5 > 2
False
```

AND

The AND logical operator is binary (i.e. takes two values). It returns True if both values are True and False otherwise. Here's its logical table :

A	*B*	*A AND B*

```
>>> True and False
False
>>> i = 0
>>> j = 1
>>> i == 0 and j > 0
True
```

OR

The OR binary logical operator returns True if at least one value is True, but returns False if both values are False.

A	*B*	*A OR B*

```
>>> True or False
True
>>> a = 5
>>> a > 1 or a < 1
True
```

XOR

The XOR binary operator (^) returns True if only one value is True and returns False otherwise.

A	*B*	*A ^B*

```
>>> a = 2
>>> a > 1 ^ a < 1
False
>>> (a > 1) ^ (a < 1)
True
```

The two last examples might seem confusing to you : it's that the logical XOR operator is the same as the binary XOR operator. I won't go into detail here ; I might get back to it. Or maybe not.