Lecture 1: Intro to Python

We use python 3 (more specifically, [python 3.2](https://docs.python.org/3.2/) on the VM, but any version above python 3.2.3 will do fine). Python 2.\* implements printing, input, integer division differently than python 3. If you see a strange error, check whether you are running python 2 or python 3.

**How to start?** At the command prompt (in the terminal window) type python3, or ipython3 (not python, which will start python 2.\*). This will start an interactive python session. We prefer to use [ipython3](http://ipython.org/ipython-doc/stable/interactive/tutorial.html), which is nicer to work with. Amongst other things, ipython gives you TAB based command completion. You can write python programs (also called "scripts") using any text editor (pretty much the same was as with a C program). To run such a python program, type

$ python3 <program>

at the command prompt ($ is part of the prompt, of course, you won't type it). Substitute <program> with the name of the program you created. By convention, python programs have an extension of .py (e.g., "hello.py").

Python is an **interpreted language**: To run a python program, you need a python interpreter (this is what you start when typing python3). Your program is compiled to the internal python byte-code which is then executed by the python interpreter, as opposed to the C language where the compiler creates machine code, which is directly fed to the CPU. Wherever the interpreter is available, your program is going to run with no need to "recompile it" for different machines.

At the interactive python session, you can type **help**() to enter python's interactive help session. Explore keywords, topics, etc. there. You can also type help(<something>) to explore a help associated with <something> (e.g., help(print)).

Further **reference material** to get you started:

* [Python 3.2 tutorial](https://docs.python.org/3.2/tutorial/)
* [Python Circles](http://cscircles.cemc.uwaterloo.ca/) is an interactive website that teaches you Python through a series of 18 lectures. You can finish this in about 10-20 hours. But this is also good for practicing.
* The CMPUT 174 slides on [eclass](https://eclass.srv.ualberta.ca/course/view.php?id=7285) from 2012-2013 can give you some more of the nitty gritty details of python.

[**Operators**](https://docs.python.org/3.2/reference/lexical_analysis.html#operators) are pretty much as in C, except that we have // (integer division), and things work differently with types: 1/3 (dividing two integers) will give you a float, whereas in C this would be integer division! (This is another difference to python 2). You can have mixed operations: "hello"\*2 duplicates the string and will give you "hellohello" (try at the command prompt). Often mixed operations do not make sense, like "hello"+2. You can **convert values** using the name of a type as in a function call. For example, "hello"+str(2) will first convert the numerical value 2 to the string "2" and then add it to the string "hello" (str is the name of the built in type string; and yes, finally we got decent support for working with strings!). Other types include: int, float, list, tuple, dict, set. We will learn about more types later (you will also learn how to create new types).

Single line **comments** are started with "#". Multiline comments are enclosed between pairs of ''' (three single apostrophes), like this:

'''This is a multiline

comment'''

All values created in Python (numeric or other) are associated with a **type**: The type is stored next the value. It is actually part of the value. Variables refer to such typed values (=objects). Since the types are stored with the values, the variables do not have types! For this reason we also say that Python is **dynamically typed**. You do not have declare variables (as the variables do not have an inherent type, unlike in C, a statically typed language), but still it does not make sense to refer to a variable before it is **defined**.  A common way of defining a new variable is using an **assignment**, which takes the form:

<identifier> = <expression\_giving\_a\_value>

Here, replace the identifier with a valid identifier (valid identifiers start with an alphabetic character or underscore, and continue with alphanumeric characters or underscore, much like in C). The identifier will give you the name of the variable. When interpreting an assignment, python first creates the value corresponding to the expression on the right-hand side of the assignment operator in the memory. This means allocating a little (or more) memory as needed, putting the type next to the value and taking the address of the start of the allocated chunk. Next it checks its **namespaces** (which essentially store name-address associations) to see whether the given name exist (in reality, the names are replaced in a precompilation phase by numbers, but the idea is the same, so let's still think that python deals with the names we write in our codes). If the name exists, its associated address is replaced by the new address of the value created. If it does not exist, a new entry is created where the address is stored against next to the name. To get the address of a value associated with an <expression>, use id(<expression>); id() is a [built-in function](https://docs.python.org/3.2/library/functions.html). For nicer formatting, use hex(id(<expression>)). Values are created dynamically in memory: When no one is referencing them, they are automatically removed: They are "garbage collected". Python keeps track of this, by associating with each value a counter that counts how many references exist in total to a given value (objects can refer to each other, as well as variables can also refer to objects). In summary, objects are: type, reference count, and value triplets.

The python interpreter won't be able to check for type compatibility of operations until the code is run (i.e., it won't warn you that "hello"+2 is illegal before it arrives at attempting to calculating the value of this expression), as opposed to the C/C++ compiler that can catch a lot of type errors at compile time (because it can check the compatibility of types participating in the various expressions). **Hence, it is important to thoroughly test your code or you will never recover from the mistakes you will make! Run your code frequently. Develop thorough test routines for every line of code you write.**

**Numeric types** include integers and floating point numbers (you also have complex numbers if you care). Integers do not have size limits (whatever fits the memory will work, but of course computation will slow down tremendously if you do computations with numbers that take many many bytes just to read). Up-promotion is automatic: 2\*1.0 will result in the float 2.0, just like 2/1, which will result in 2.0 (if you want integer results, use 2//1). Clock arithmetics is supported using %. **Floating point numbers** are essentially a pair of an integer mantissa and and exponent; basically a floating point number is mantissa\*(2\*\*exponent), where mantissa will have a certain number of bits (53 in most python implementations) and the exponent will have in general fewer bits (10). Here, 10+53 = 63, 1 bit is reserved to store the sign (so the mantissa is not stored in 2s complement). You can read about them [here](http://cs.iupui.edu/~aharris/230/twosCompFloat.html) (very nice overview), or on the [python doc website](https://docs.python.org/3.2/tutorial/floatingpoint.html), or on [Wikipedia](http://en.wikipedia.org/wiki/Floating_point). The support for floating point is as good as it gets: But since floating point numbers can only represent a small subset of real numbers, you should not expect real number operations to work without errors. For example, 0.1+0.1+0.1 will be different from 0.3 (check this at the command prompt).

Besides assignment statements, which are basically one line statement, we learned about compound statements to support **iteration** and **decision making**. The associated keywords for the two types of loops are [**for**](https://docs.python.org/3.2/reference/compound_stmts.html#the-for-statement) and [**while**](https://docs.python.org/3.2/reference/compound_stmts.html#the-while-statement), and for decision making you have [**if**](https://docs.python.org/3.2/reference/compound_stmts.html#the-if-statement), **else**, **elif**. For compound statements, python uses indentation to indicate the beginning and end of a **block** (also called a suite in python lingo) associated with the compound statement. Use TAB for indentation, do not even think of using spaces, not mentioning mixing spaces and tabs! The syntax of if is:

if <logical\_expression>:

<statement1>

<statement2>

...

...

Here, the second ... shows the statement that is outside of the "scope" (in a C sense) of the if compound statement. We talked about True and False, the built in constants to represent the logical value of true and false.

The nice thing about for is that it can **loop through** any [**sequence types**](https://docs.python.org/3.2/library/stdtypes.html#typesseq), like strings, or lists in a native way:

for <variable> in <expression\_of\_sequence\_type>:

<suite>

will do no matter what the actual sequence type is. The syntax for a while loop is:

while <logical\_expression>: <suite>

You can use the [range](https://docs.python.org/3.2/library/functions.html?highlight=range#range)() function to **iterate through a list of integers** arranged in an arithmetic progression.

As opposed to C, [strings](https://docs.python.org/3.2/library/functions.html#str) are finally first class citizens. A string literal can be constructed like in C using "example", but we can also use single quotation marks: 'another example'. There is a rich set of operations on strings (see the previous link), amongst which we talked about string formatting, which roughly allows you do things similar to what you can do in C with printf. The best way to learn about these is to look at some [examples](https://docs.python.org/3.2/library/string.html#format-examples).

In python, [lists](https://docs.python.org/3.2/library/functions.html?highlight=range#list) are like C arrays, just they can hold any type of data (actually they hold addresses to objects) and their size can be dynamically changed. To create a list holding the numeric values 1,2,3 (as an example), you can use the expression [1,2,3] (that is, the values to be put into the list in a bracket). You can make all kind of operations on lists -- we'll cover these later.

To access some character at some position i in a string s, use s[i]. This works for lists, too: If lst holds a reference to a list object, lst[i] will give you the i-th item on the list.

To get the length of a sequence use the function [len](https://docs.python.org/3.2/library/functions.html?highlight=range#len)(), like len("adios"), or len(["a",2,"b"]).

In addition to the built in function len(), we discussed [input](https://docs.python.org/3.2/library/functions.html?highlight=range#input)(<prompt>) which writes a prompt to the screen and asks the user to enter a value which is returned as a string, and [print](https://docs.python.org/3.2/library/functions.html?highlight=range#print), which allows you to print things to the screen.