1 Practical Bioinformatics – Day 1

1.1 Saying hello to the interpreter shell

```
print "hello, world"
```

```
hello, world
```

Commands that start with "%" are IPython "magics", which are not part of the python language and do not get sent directly to the interpreter.

You can get a list of available magics with:

`%magic`

```
%pwd
```

```
u'/home/mvoorhie/BMS270'
```

Commands that start with "!" are sent directly to the operating system shell rather than to the python interpreter.

On OS X and Linux, the shell is typically bash, which will interpret `!ls` as *list the files in the current directory*.

On Windows, `!dir` should give the equivalent result in the DOS shell (if you start ipython from cygwin, you should be able to use bash commands).

```
!ls
```

```
notebook1.ipynb  notebook1.log
```

Here is a platform independent way to have python create a list of the filenames in the current directory

```
import glob
glob.glob("*")
```

```
['notebook1.log', 'notebook1.ipynb']
```

The %logstart magic retro-actively starts logging this IPython session (*i.e.*, it captures even commands typed before the log was started).

The -o option turns on logging for outputs in addition to inputs.

```
%logstart -o notebook1.log
```

```
Activating auto-logging. Current session state plus future input saved.
Filename       : notebook1.log
Mode           : backup
Output logging : True
Raw input log  : False
Timestamping   : False
State          : active
```

1.2 Data types

Use matched single or double quotes for simple strings, triple quotes for multi-line strings.

```
"this is a string"
```

```
'this is a string'
```

```
'i can single quote'
```

```
'i can single quote'
```

```
"""this won't do what I want

multiple

                lines
"""
```

```
"this won't do what I want\n\nmultiple\n\n                lines\n"
```

```
print """this won't do what I want

multiple

                lines
"""
```

```
this won't do what I want

multiple

                 lines
```

Use a decimal point to distinguish integer (int) and double-precision floating point (oat) values.

```
5
```

```
5
```

```
5.
```

```
5.0
```

```
5+3
```

```
8
```

```
5-3
```

```
2
```

```
5*2
```

```
10
```

```
10/2
```

```
5
```

Dividing two integers gives an integer result, rounded down.

```
10/3
```

```
3
```

Make sure to include at least one float if you want floating point division

```
10./3.
```

```
3.3333333333333335
```

```
10/3.
```

```
3.3333333333333335
```

String substitution works via the mod (`%`) operator

```
"This is a %s mad lib" % "neat"
```

```
'This is a neat mad lib'
```

Use parenthesis to group multiple arguments for string substitution
Technically, the parenthesis are defining a tuple, which is an immutable version of a list

```
"Now %s and %s" % ("one","two")
```

```
'Now one and two'
```

Most python objects can be converted to strings, which is what the `%s` format string is asking for

```
"Now %s and %s" % (1,2)
```

```
'Now 1 and 2'
```

```
"%s" % (10./3)
```

```
'3.33333333333'
```

The `%f` format string lets us specify special formatting for floating point values; *e.g.*, the number of decimal places:

```
"%.5f" % (10./3)
```

```
'3.33333'
```

```
"%2.5f" % (10./3)
```

```
'3.33333'
```

The `%e` format string gives scientific notation:

```
"%e" % (10./3)
```

```
'3.333333e+00'
```

```
2**3
```

```
8
```

1.3 Assignment

The assignment operator (`=`) lets a name "point at" (reference) a piece of data.

We can refer to the names as *variables* (because their value may vary with reassignment) or as *references* (to emphasize their pointing nature).

```
a = 8
b = "hello"
```

We can then operate on the references as if they were the underlying data

```
a
```

```
8
```

```
b
```

```
'hello'
```

```
a + 3
```

```
11
```

```
a + b
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-31-f96fb8f649b6> in <module>()
----> 1 a + b

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Addition on strings is concatenation:

```
b + "hi"
```

```
'hellohi'
```

```
a
```

```
8
```

```
a = a + 3
```

```
a
```

```
11
```

```
a += 3
```

```
a
```

```
14
```

```
(a+2)*3
```

```
48
```

Exercise: using variables to evaluate a formula

```
ngc = 13
L = 25
nmm = 2

# Wrong: gives an unintended rounding error due to integer division
Tm1 = 81.5 + (41*ngc - 100*nmm - 675)/L
# Here are two ways to ensure floating point division:
Tm2 = 81.5 + (41*ngc - 100*nmm - 675.)/L
Tm3 = 81.5 + (41*ngc - 100*nmm - 675)/float(L)
```

```
Tm1, Tm2, Tm3
```

```
(67.5, 67.82, 67.82)
```

IPython magic for listing just the names that we've explicitly assigned in this session

```
%who
```

```
L Tm1 Tm2 Tm3 a b glob ngc nmm
```

How to remove a reference:

```
del Tm1
```

```
%who
```

```
L Tm2 Tm3 a b glob ngc nmm
```

```
a
```

```
14
```

1.4 Comparison operators

`=` → assignment
`==` → test for equality

```
a == 5
```

```
False
```

```
a = 5
```

```
a == 5
```

```
True
```

```
b = True
```

```
a > 6
```

```
False
```

```
a < 5
```

```
False
```

```
a != 3
```

```
True
```

```
a >= 3
```

```
True
```

```
a <= 3
```

```
False
```

Boolean values can be combined with **and**, **or**, and **not**.

Remember that this type of logical expression will "short circuit" as soon as only one outcome is possible.

```
(5 > 3) and (3 > 2)
```

```
True
```

```
(5 < 3) or (3 > 2)
```

```
True
```

We can act on boolean values using if/elif/else statements.

Things to remember for this type of block statement:

- First part ends with a colon (:)
- The block of code that the first part refers to is indicated by indentation – everything at the same indent level is part of the same code block.

```
if(5 > 6):
    print "it is"
else:
    print "it isn't"
```

```
it isn't
```

```
if(5 > 6):
    print "it is"
else:
    print "it isn't"
    print "it really isn't"
```

```
it isn't
it really isn't
```

```
if(5 > 6):
    print "hi"
elif(5> 3):
    print "hello"
else:
    print "the end"
```

```
hello
```

```
if(5 > 3):
    if(4 > 7):
        print "first"
    else:
        print "second"
else:
    print "oops"
```

```
second
```

1.5 Lists

Lists are:

- Indicated by square brackets (`[]`)
- Heterogenous (can contain more than one type of data)
- Dynamic (can be grown with `+`, `+=`, and `append`, and can have elements reassigned)
- Indexed from zero

```
mylist = [1,2,"apple","orange",2**5]
```

```
mylist
```

```
[1, 2, 'apple', 'orange', 32]
```

```
mylist += ["first","second"]
```

```
mylist
```

```
[1, 2, 'apple', 'orange', 32, 'first', 'second']
```

```
mylist.append(5)
```

```
mylist
```

```
[1, 2, 'apple', 'orange', 32, 'first', 'second', 5]
```

```
list2 = []
```

```
list2.append(1)
```

```
list2
```

```
[1]
```

```
mylist[3]
```

```
'orange'
```

```
mylist[3:6]
```

```
['orange', 32, 'first']
```

We can find the length of a sequence or collection with len
When defining your own classes, you can add this property with a __len__ method

```
len(mylist)
```

```
8
```

```
len("string")
```

```
6
```

```
len(6)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-78-d42e7c5a4468> in <module>()
----> 1 len(6)

TypeError: object of type 'int' has no len()
```

```
mylist[len(mylist)-1]
```

```
5
```

```
mylist[-1]
```

```
5
```

```
mylist[:3]
```

```
[1, 2, 'apple']
```

```
mylist[3:]
```

```
['orange', 32, 'first', 'second', 5]
```

```
mylist[:]
```

```
[1, 2, 'apple', 'orange', 32, 'first', 'second', 5]
```

The "pointing at" nature of python references can give unexpected behavior for lists. Use `x[:]` to make a "shallow" copy of a list for independent modification.

```
otherlist = mylist
```

```
print otherlist
print mylist
```

```
[1, 2, 'apple', 'orange', 32, 'first', 'second', 5]
[1, 2, 'apple', 'orange', 32, 'first', 'second', 5]
```

```
otherlist[3] = "thing"
```

```
otherlist
```

```
[1, 2, 'apple', 'thing', 32, 'first', 'second', 5]
```

```
mylist
```

```
[1, 2, 'apple', 'thing', 32, 'first', 'second', 5]
```

```
otherlist = mylist[:]
```

```
otherlist[3] = "new"
```

```
print mylist
print otherlist
```

```
[1, 2, 'apple', 'thing', 32, 'first', 'second', 5]
[1, 2, 'apple', 'new', 32, 'first', 'second', 5]
```

```
c = mylist[:3]
```

```
c[2] = 5
```

```
print c
print mylist
```

```
[1, 2, 5]
[1, 2, 'apple', 'thing', 32, 'first', 'second', 5]
```

We can create a (possibly ragged) multi-dimensional array with a list of lists:

```
L = [[5,6],[3,4]]
```

```
L
```

```
[[5, 6], [3, 4]]
```

```
L[0][1]
```

```
6
```

We can also create multidimensional arrays with the numpy `array` class.
Restrictions on arrays:

- Homogenous (all data must be of the same type, which can be specified with `dtype`)
- Not ragged (all rows must be the same length)
- Not dynamic (dimensions are fixed at the time of creation; however, an array may be reshaped, and values within the array can be modified).

Benifits of arrays (due to underlying C implementation):

- More memory-efficient than python lists
- Faster math via numpy/scipy/etc

For starting out, you'll usually want python lists

```
A = array(L)
```

```
A
```

```
array([[5, 6],
       [3, 4]])
```