1 Practical Bioinformatics – Day 2

1.1 Definining functions

De ning a function for the mean calculation:

```
def mean(x):
    """Return the mean of a list of numbers."""
    s = 0.
    for i in x:
        s += i
    return s/len(x)
```

```
L1 = [.5,10,17,3]
```

```
mean(L1)
```

```
7.625
```

Should we report our result with **return** or **print**?

```
def mean2(x):
    """Print the mean of a list of numbers."""
    s = 0.
    for i in x:
        s += i
    print s/len(x)
```

```
x = mean(L1)
y = mean2(L1)
```

```
7.625
```

mean uses **return**, so the result is stored in **x**.

```
x
```

```
7.625
```

mean2 uses **print** with no **return** statement, so it returns the default return value, which is **None**.

```
y
```

```
z = None
```

```
z
```

Normal test for equality: **z** and **y** both point to **None**, so their values are equal.

```
z == y
```

```
True
```

Test for identity: **z** and **y** point to the same location in memory (because the **None** value is unique and lives in a single location)

```
z is y
```

```
True
```

Advanced exercise: The **"%p"** format string returns an object's address in memory. Can you use this to gure out how python handles memory management for **int** and **float** values. Is there a di erence?

Should we document our function with a string or a comment?

```
def mean3(x):
    #Return the mean of a list44.56RG0g0G.26.44.56rg.26.44.56RG[-525(mean)]T 0g0G.26.44.56rg.26.44.56RG
    s = 0.
    for i in x:
        s += i
    return s/len(x)
```

Because **mean** is documented with a docstring, its documentation is visible to IPython via **mean?** or tab-completion on **mean(**.

mean3 uses a comment, which is invisible to the built in help.

Functions for creating integer sequences:

- **range(n)**

```
1000 loops, best of 3: 570 µs per loop
```

```
%time?
```

Demonstrating that IPython magics are converted to python function calls when logging:

```
%logstart -o day2.log
```

```
Activating auto-logging. Current session state plus future input saved.
Filename       : day2.log
Mode           : backup
Output logging : True
Raw input log  : False
Timestamping   : False
State          : active
```

```
%less day2.log
```

Using introspection to find the source of the **sqrt** function in the pylab environment:

```
sqrt.__class__
```

```
numpy.ufunc
```

Ah, it's the version from **numpy** (not part of standard Python. Standard Python includes a slightly different **sqrt** implementation in the **math** module)

```
sqrt.__class__.__module__
```

```
'numpy'
```

? is an IPython-specific way to look up the documentation and source for an object:

```
sqrt?
```

1.2 Importing functions from external modules

- Module filenames end with *.py"
- Place your modules in the same directory that you launch IPython from, for ease of import
- Use **import** until a module loads successfully
- After a successful load, use **reload** to force Python to re-read the contents of the \.py" file.

Advanced: There are several ways to import modules from directories other than the current directory. The most common is to modify the PYTHONPATH environment variable. You can inspect and modify the current import path via sys.path.

```
import stats
```

```
stats?
```

dir lists the contents of an object:

```
dir(stats)
```

```
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'mean']
```

```
stats.mean?
```

```
stats.mean(L1)
```

```
7.625
```

```
from stats import mean
```

```
mean(L1)
```

```
7.625
```

Here, we edited the de nition of **mean** in stats.py and experimented with how to make the change visible to Python:

```
mean(L1)
```

```
7.625
```

```
stats.mean(L1)
```

```
7.625
```

```
import stats
```

```
stats.mean(L1)
```

```
7.625
```

```
reload(stats)
```

```
<module 'stats' from 'stats.py'>
```

```
stats.mean(L1)
```

```
'Oops!'
```

```
mean(L1)
```

```
7.625
```

```
from stats import mean
```

```
mean(L1)
```

```
'Oops!'
```

1.3 File I/O

Opening a le in read-only mode:

```
fp = open("stats.py")
```

Read the rst 10 bytes:

Python scripts and modules are simple text les, so Python translates the bytes to text using its default text encoding

```
fp.read(10)
```

```
'"""Some de'
```

Read the next 10 bytes (note that read has a *side e ect*: the le pointer moves through the le as it is read, so sequential calls to read produce di erent results)

```
fp.read(10)
```

```
'scriptive '
```

Read the rest of the le:

```
fp.read()
```

```
'statistics"""\n\ndef mean(x):\n    """Return the mean of a list of numbers."""\n    s = 0.\n    for i i
```

Rewind by reopening the le

```
fp = open("stats.py")
```

Because stats.py is a text le, we can use **readline** in place of **read** to read a whole line of text (including the end-of-line character)

```
fp.readline()
```

```
'"""Some descriptive statistics"""\n'
```

```
fp.readline()
```

```
'\n'
```

```
fp.readline()
```

```
'def mean(x):\n'
```

For **file** objects, **next** has the same behavior as **readline**.

```
fp.next()
```

```
'    """Return the mean of a list of numbers."""\n'
```

Objects with a **next** method are *iterators*, which can be used by a **for** loop

To be more precise: **for** loops operate on **iterables**, which are objects with an **iter** method, which is responsible for generating or pointing to the appropriate **iterator** object

```
for i in open("stats.py"):
    print i
```

```
"""Some descriptive statistics"""



def mean(x):

    """Return the mean of a list of numbers."""

    s = 0.

    for i in x:

        s += i

    #return s/len(x)

    return "Oops!"
```

The example above is double spaced, because **print** adds its own newline (*n*) character. We can use the **rstrip** string method to remove the redundant newline (and any other trailing whitespace) from each line of the le:

```
for i in open("stats.py"):
    print i.rstrip()
```

```
"""Some descriptive statistics"""

def mean(x):
    """Return the mean of a list of numbers."""
    s = 0.
    for i in x:
        s += i
    #return s/len(x)
    return "Oops!"
```

To write data to disk, we can open a file in write mode:

```
out = open("example.txt","w")
```

```
out.write("This is some text")
out.write("this is some other text")
out.close()
```

```
for line in open("example.txt"):
    print line.rstrip()
```

```
This is some textthis is some other text
```

If we want to write output to separate lines, we need to add explicit newlines:

```
out = open("example.txt","w")
out.write("This is some text\n")
out.write("this is some other text\n")
out.close()
```

```
for line in open("example.txt"):
    print line.rstrip()
```

```
This is some text
this is some other text
```

We have been using relative paths, which look for files relative to the current directory. We can specify absolute paths instead, by starting with a forward slash (/).

Windows uses backslash (\) as its default path separator, but Python will do the correct conversion of forward slash on Windows, so you don't need to worry about this

```
for line in open("/home/mvoorhie/Projects/Courses/PracticalBioinformatics/python/May2013/example.txt"):
    print line.rstrip()
```

```
This is some text
this is some other text
```

1.4 Inspecting the example tab-delimited expression data from the PNAS paper

readlines returns all lines of a text le as a list of strings

```
data = open("supp2data.cdt").readlines()
```

```
len(data)
```

```
2468
```

```
data[0]
```

```
'ORF\tNAME\talpha 0\talpha 7\talpha 14\talpha 21\talpha 28\talpha 35\talpha 42\talpha 49\talpha 56\talph
```

```
print data[0]
```

```
ORF NAME alpha 0 alpha 7 alpha 14 alpha 21 alpha 28 alpha 35 alpha 42 alpha 49 alpha 56
alpha 63 alpha 70 alpha 77 alpha 84 alpha 91 alpha 98 alpha 105 alpha 112 alpha 119
Elu 0 Elu 30 Elu 60 Elu 90 Elu 120 Elu 150 Elu 180 Elu 210 Elu 240 Elu 270 Elu 300
Elu 330 Elu 360 Elu 390 cdc15 10 cdc15 30 cdc15 50 cdc15 70 cdc15 90 cdc15 110 cdc15 130
cdc15 150 cdc15 170 cdc15 190 cdc15 210 cdc15 230 cdc15 250 cdc15 270 cdc15 290 spo 0
spo 2 spo 5 spo 7 spo 9 spo 11 spo5 2 spo5 7 spo5 11 spo- early spo- mid heat 0 heat 10
heat 20 heat 40 heat 80 heat 160 dtt 15 dtt 30 dtt 60 dtt 120 cold 0 cold 20 cold 40
cold 160 diau a diau b diau c diau d diau e diau f diau g
```

```
data[1]
```

```
'YBR166C\tTYR1   TYROSINE BIOSYNTHESIS    PREPHENATE DEHYDROGENASE (NADP+\t0.33\t-0.17\t0.04\t-0.07\t-0.
```

```
data[-1]
```

```
'YLR160C\tASP3   ASPARAGINE UTILIZATION   L-ASPARAGINASE II\t0.07\t-0.04\t0.12\t-0.1\t0.06\t-0.32\t0.21\
```

The string **split** method splits a line on whitespace (or on an arbitrary string, if supplied)

```
data[1].split()
```

```
['YBR166C',
 'TYR1',
 'TYROSINE',
 'BIOSYNTHESIS',
 'PREPHENATE',
 'DEHYDROGENASE',
 '(NADP+',
 '0.33',
 '-0.17',
 '0.04',
 '-0.07',
 '-0.09',
```

```
'-0.12',
'-0.03',
'-0.2',
'-0.06',
'-0.06',
'-0.14',
'-0.18',
'-0.06',
'-0.25',
'0.06',
'-0.12',
'0.25',
'0.43',
'0.21',
'-0.04',
'-0.15',
'-0.04',
'0.21',
'-0.14',
'-0.03',
'-0.07',
'-0.36',
'-0.14',
'-0.42',
'-0.34',
'-0.23',
'-0.17',
'0.23',
'0.3',
'0.41',
'-0.07',
'-0.23',
'-0.12',
'0.16',
'0.74',
'0.14',
'-0.49',
'-0.32',
'0.19',
'0.23',
'0.24',
'0.28',
'1.13',
'-0.12',
'0.1',
'0.66',
'0.62',
'0.08',
'0.62',
'0.43',
'0.5',
'-0.25',
```

```
 '-0.51',
 '-0.67',
 '0.21',
 '-0.74',
 '-0.36',
 '-0.01',
 '0.38',
 '0.15',
 '-0.22',
 '-0.09',
 '0.33',
 '0.08',
 '0.39',
 '-0.17',
 '0.23',
 '0.2',
 '0.2',
 '-0.17',
 '-0.69',
 '0.14',
 '-0.27']
```

Here's how to split on tabs:

```
data[1].split("\t")[:10]
```

```
['YBR166C',
 'TYR1   TYROSINE BIOSYNTHESIS    PREPHENATE DEHYDROGENASE (NADP+',
 '0.33',
 '-0.17',
 '0.04',
 '-0.07',
 '-0.09',
 '-0.12',
 '-0.03',
 '-0.2']
```

```
data[1].split("\t")[2:10]
```

```
['0.33', '-0.17', '0.04', '-0.07', '-0.09', '-0.12', '-0.03', '-0.2']
```

Coercing strings to floats:

```
float(data[1].split("\t")[2])
```

```
0.33
```

Handling missing values with a **try..except** block:

```
r = []
for i in data[1].split("\t")[2:]:
```

```
    try:
        r.append(float(i))
    except ValueError:
        print i
```

```
r = []
for i in data[1].split("\t")[2:]:
    try:
        r.append(float(i))
    except ValueError:
        r.append(None)
```

```
print r
```

```
[0.33, -0.17, 0.04, -0.07, -0.09, -0.12, -0.03, -0.2, -0.06, -0.06, -0.14, -0.18, -0.06, -0.25, 0.06, -0
```

```
r = []
for i in data[1].split("\t")[2:]:
    try:
        r.append(float(i))
    except ValueError:
        r.append(0.)
```

```
print r
```

```
[0.33, -0.17, 0.04, -0.07, -0.09, -0.12, -0.03, -0.2, -0.06, -0.06, -0.14, -0.18, -0.06, -0.25, 0.06, -0
```

Oops! Better log my session!

```
%logstart -o BMS270b.2013.02.log
```

```
Activating auto-logging. Current session state plus future input saved.
Filename       : BMS270b.2013.02.log
Mode           : backup
Output logging : True
Raw input log  : False
Timestamping   : False
State          : active
```