# Contents



# 1 python

This is mostly based on the official python tutorial: reading the tutorial is highly recommended to make sure your python is up to speed.

## 1.1 Data Structures: lists, tuples, dicts

- list is just that: a list of things
  - technically they are instances of classes, like everything in python

- a string is also a list, an *immutable* list of characters
- list is agnostic to its contents

```
lista = [1,2,3]
listb = "I am a string"
listc = [1, "a", lambda x:x+1]
lista[2] = 42
print(listc[2](3))
```

- but strings were immutable, ok?

```
listb[0] = "x"
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

- lists can be appended to
  - but prepending is very inefficient, do not do it!
  - use `collections.deque` if you need to add to both ends
- lists can also be "added" together (note that `lista+5` would not work as `5` is not a list

```
lista.append(4)
lista = lista+[5]
print(lista)
```

- Other useful features are operator overloading: `2*3+1==7`, but

```
'little bunny'+2*' foo'=='little bunny foo foo'
```

- we already saw with our little bunny how lists can be multiplied; division and subtraction make no sense to lists, though
  - there's also the `set` type, where subtraction does the obvious
  - sets also know intersections, unions etc, but on the other hand they have no `+` or `*`
- a tuple is like an immutable list

- and as such can be used as an index or key (mutables cannot)
- the final essential built-in type is `dict`, short for dictionary
  - used to create pairs of `key` and `value`
  - keys must be immutable
  - values can be anything
  - keys are *unsorted*
  - allows quick addition and removal of elements in a dict

```
dictionary = {"key1": "value1",
              "key2": "value2"}
loci = {(1,2): "person 1",
        (2,42): "person 2",
        (32,8): "person 3"}
for k,v in dictionary.items():
    print('key "{k}" has value "{v}"'.format(k=k,v=v))
coord=(2,42)
print("It is {cmp} that there is a person at {coord}.".format(
    cmp=coord in loci.keys(), coord=coord))
b=(1,42)
print("Equality of {a} and {b} evaluates to {val}.".format(
    a=coord, b=b, val=b==coord))
```

```
key "key1" has value "value1"
key "key2" has value "value2"
It is True that there is a person at (2, 42).
Equality of (2, 42) and (1, 42) evaluates to False.
```

## 1.2 Slicing of arrays and strings

This will be used heavily, so a few interactive examples

```
x="string"
x
```

```
string
```

```
x[0]
```

```
s
```

```
x[-1]
```

```
g
```

```
x[2:-2]
```

```
ri
```

### 1.3 numpy has some extra slicing features

- For numerical work in python, you should nearly always use the `numpy` package. We'll cover packages later, but for now the numpy package can be used after the command `import numpy`.
- First, let's see what a numpy array looks like: this is a 2x3x4 array.

```
import numpy
skewed = numpy.random.random((2,3,4))
skewed
```

| | | |
|---|---|---|
| (0.81099375 0.91880462 0.62830062 0.28378392) | (0.20960746 0.52254401 0.97153996 0.09809617) | ( |
| (0.76978011 0.89910868 0.16330815 0.92383601) | (0.86548926 0.32243773 0.0829573 0.32117172) | ( |

We will use `array` in our examples later, we will initially create it as follows

```
array = numpy.arange(0,27)
array
```

| | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |

but now we give it a new shape

```
array=array.reshape(3,3,3)
array
```

| | | |
|---|---|---|
| (0 1 2) | (3 4 5) | (6 7 8) |
| (9 10 11) | (12 13 14) | (15 16 17) |
| (18 19 20) | (21 22 23) | (24 25 26) |

- this is a slice

```
array[-1,:,1:]
```

| | |
|---|---|
| 19 | 20 |
| 22 | 23 |
| 25 | 26 |

- and striding is also possible, and combining the two:

```
array[::2,:,2:0:-1]
```

| | | |
|---|---|---|
| (2 1) | (5 4) | (8 7) |
| (20 19) | (23 22) | (26 25) |

- numpy arrays have point-wise operator overloads:

```
array + array
```

| | | |
|---|---|---|
| (0 2 4) | (6 8 10) | (12 14 16) |
| (18 20 22) | (24 26 28) | (30 32 34) |
| (36 38 40) | (42 44 46) | (48 50 52) |

```
array * array
```

| | | |
|---|---|---|
| (0 1 4) | (9 16 25) | (36 49 64) |
| (81 100 121) | (144 169 196) | (225 256 289) |
| (324 361 400) | (441 484 529) | (576 625 676) |

- nearly any imaginable mathematical function can operate on an array element-wise:

```
array ** 0.5
```

| | | |
|---|---|---|
| (0 1 1.41421356) | (1.73205081 2 2.23606798) | (2.44948974 2.64575131 |
| (3 3.16227766 3.31662479) | (3.46410162 3.60555128 3.74165739) | (3.87298335 4 4.1231056 |
| (4.24264069 4.35889894 4.47213595) | (4.58257569 4.69041576 4.79583152) | (4.89897949 5 5.0990195 |

```
numpy.sinh(array)
```

| | | |
|---|---|---|
| (0.0 1.17520119 3.62686041) | (10.0178749 27.2899172 74.2032106) | (201.713157 548.3 |
| (4051.5419 11013.2329 29937.0708) | (81377.3957 221206.696 601302.142) | (1634508.69 4443 |
| (32829984.6 89241150.5 242582598.0) | (659407867.0 1792456420.0 4872401720.0) | (13244561100.0 3 |

- but numpy matrices are bona fide matrices:

```
matrix=numpy.matrix(array[0,:,:])
matrix*matrix
```

| | | |
|---|---|---|
| 15 | 18 | 21 |
| 42 | 54 | 66 |
| 69 | 90 | 111 |

- numpy matrices have all the basic operations defined, but not necessarily with good performance
- for prototyping they're fine
- **performance can be exceptional if numpy compiled suitably**
- if you import `scipy` you have even more functions

```
import scipy
import scipy.special
scipy.special.kn(2,array)
```

| | |
|---|---|
| (nan 1.6248389 0.253759755) | (0.0615104585 0.0174014255 0.00530894371) |
| (6.28006499e-05 2.1509817e-05 7.42863158e-06) | (2.58261831e-06 9.02740362e-07 3.16970563e-07) |
| (4.97889218e-09 1.77354149e-09 6.32954361e-10) | (2.26274885e-10 8.1013499e-11 2.9044956e-11) |

- I should say that

```
import scipy.fftpack
scipy.fftpack.fftn(array)
```

| | | |
|---|---|---|
| (351.0 +0.j -13.5 +7.79422863j -13.5 -7.79422863j) | (-40.5+23.3826859j 0.0 +0.j 0.0 +0.j) | (-40.5-23 |
| (-121.5+70.14805771j 0.0 +0.j 0.0 +0.j) | (0.0 +0.j 0.0 +0.j 0.0 +0.j) | (0.0 +0. |
| (-121.5-70.14805771j 0.0 +0.j 0.0 +0.j) | (0.0 +0.j 0.0 +0.j 0.0 +0.j) | (0.0 +0. |

- performance of the FFT routines also depends on how everytihng was compiled
- and theoretical physicists may find it amusing that numpy can do Einstein summation (and more)

```
numpy.einsum("iii", array)
```

```
39
```

```
numpy.einsum("ij,jk", array[0,:,:], array[1,:,:])
```

| | | |
|---|---|---|
| 42 | 45 | 48 |
| 150 | 162 | 174 |
| 258 | 279 | 300 |

```
numpy.einsum("ijk,ljm", array, array)
```

| | | |
|---|---|---|
| ((45 54 63) (126 135 144) (207 216 225)) | ((54 66 78) (162 174 186) (270 282 294)) | ( |
| ((126 162 198) (450 486 522) (774 810 846)) | ((135 174 213) (486 525 564) (837 876 915)) | ( |
| ((207 270 333) (774 837 900) (1341 1404 1467)) | ((216 282 348) (810 876 942) (1404 1470 1536)) | ( |

### 1.4 Control flow statements

```
if (1>0):
  print("1 is indeed greater than 0")
elif (1==0):
  print("Somehow 1 is equal to 0 now")
else:
  print("Weird, 1 is somehow less than 0!")
```

```
1 is indeed greater than 0
```

```
for i in [0,1,2,3]:
  print(str(i))
```

```
0
1
2
3
```

```
for i in range(4):
  print(str(i))
```

```
0
1
2
3
```

```
for i in range(4):
  print(str(i), end="")
```

```
0123
```

```
for i in range(0,4): print(str(i), end=", ")
```

```
0, 1, 2, 3,
```

```
print([i for i in range(0,4)])
```

```
[0, 1, 2, 3]
```

```
print([str(i) for i in range(0,4)])
```

```
['0', '1', '2', '3']
```

```
for i in range(4): print(str(i), end=", ")
```

```
0, 1, 2, 3,
```

```
print(','.join([str(i) for i in range(0,4)]))
```

```
0,1,2,3
```

- there are others, see the tutorial for python 3

### 1.4.1 Functions

- two types of functions: "normal" and *class methods*
- syntax is the same; we'll deal with class methods' peculiarities in a moment

```
def findzeros(a, b, c):
    '''Find the real root(s) of "a x^2 + b x + c".
    >>> findzeros(1,4,3)
    (-1.0, -3.0)
    >>> findzeros(1,2,-3)
    (1.0, -3.0)
    >>> findzeros(1,-2,-3)
    (3.0, -1.0)
    >>> findzeros(1,-4,3)
```

```
    (3.0, 1.0)
    >>> findzeros(1,0,9)[0]+3,findzeros(1,0,9)[1]+3
    ((3+3j), (3-3j))
    >>> findzeros(2,8,6)
    (-1.0, -3.0)
    >>> findzeros(1,-2,1)
    (1.0, 1.0)
    '''
    root1 = (-b + (b**2 - 4 * a * c)**0.5)/(2*a)
    root2 = (-b - (b**2 - 4 * a * c)**0.5)/(2*a)
    return (root1,root2)
```

- if the first line after the function definition is a string or multiline string, like here, it will become the function's *docstring*
  - this is a very good way of documenting your functions
  - you should rarely need other documentation in a function: it is likely too complex or long if you feel you need comments inside it
  - sometimes a clever algorithmic trick or implementation requires further comments
  - a docstring is also viewable with `help(findzeros)` or

```
print(findzeros.__doc__)
```

- a function can have *default values* for its parameters (unlike C/Fortran)

```
import urllib
import urllib.request
def get_url(url='http://www.cam.ac.uk/'):
    data=[]
    with urllib.request.urlopen(url) as response:
        charset = response.headers.get_content_charset()
        for line in response:
            data.append(line.decode(charset))
    return data
```

- just to show this works (the IPython bits are jupyter/IPython special modules, disregard for now)

```
from IPython.display import display, HTML
chart = HTML("".join(get_url()))
display(chart)
```

```
In : <IPython.core.display.HTML object>
```

- functions can have arbitrary argument lists, too
- the name `args` is not special, just a convention

```
def multiply(*args):
    res=1
    for a in args:
        res = res*a
    return res
print(multiply())
print(multiply(1))
print(multiply(1,2))
print(multiply(1,2,3))
print(multiply(42,42))
```

```
1
1
2
6
1764
```

- and also unspecified *keyword arguments* which become a dict inside the function
- in fact in the previous example, one could call `get_url("http://www.python.org")` without the `url=` part: not so with `**kwargs`

```
def func_with_kwargs(**kwargs):
    for key in kwargs:
        print("The key {key:20} has the value {value:20}.".format(
            key=key, value=kwargs[key]))
    return
```

- these are used just as `url` above (it is actually also a keyword argument just a named one)

```
func_with_kwargs(foo=8, bar="9", foobar=89)
```

```
The key foo                   has the value                    8.
The key bar                   has the value 9                   .
The key foobar                has the value                   89.
```

- note how the alignment of strings and numbers is different in `print`!
- a function can mix and match all types of arguments, but
  - but order matters in definition: the following function has all types of arguments and the order of the types of arguments is the only allowed one
  - order also matters when calling: pay attention to the numbers 5 and 6 in the example

```
def many_args(a, b, c=42, d=0, *e, **f):
    print("a = "+str(a))
    print("b = "+str(b))
    print("c = "+str(c))
    print("d = "+str(d))
    for i,E in enumerate(e): print("e[{idx}] = ".format(idx=i) + str(E))
    for F in f: print("f[{key}] = ".format(key=F) + str(f[F]))

many_args(1, 2, 3, 4, 5, 6, bar=8)
many_args(1, 2, d=3, c=4, bar=8)
```

```
a = 1
b = 2
c = 3
d = 4
e[0] = 5
e[1] = 6
f[bar] = 8
a = 1
b = 2
c = 4
d = 3
f[bar] = 8
```

- note how `c` and `d` can be passed in any order
- but these do not work

```
many_args(1, 2, d=3, c=4, 6, bar=8)
```

```
File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

```
many_args(1, 2, 6, d=3, c=4, bar=8)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: many_args() got multiple values for argument 'c'
```

- so cannot pass `*args` style parameters and named keyword arguments at the same time except in the right order
- but then again, the unnamed keyword arguments can be mixed with the named ones

```
many_args(1, 2, bar=8, d=3, c=4)
```

```
a = 1
b = 2
c = 4
d = 3
f[bar] = 8
```

### 1.4.2 Anonymous Function

- also known as a *lambda*
- a breeze from the land of functional programming, very useful
- especially with `reduce` and `map` operating on lists

```
from __future__ import print_function
list_of_values = ["a", "b", "c", "abc"]
uppercase_list_of_values = map(lambda x: x.upper(), list_of_values)
print("uppercase_list_of_values = ",end="")
print("".join(uppercase_list_of_values))
uppercase_list_of_values_without_lambda = [x.upper() for x in list_of_values]
print("uppercase_list_of_values_without_lambda = ",end="")
print(uppercase_list_of_values_without_lambda)
import functools
joined_list_of_values = functools.reduce(lambda x,y: x+y, list_of_values, "")
print("joined_list_of_values = ",end="")
print(joined_list_of_values)
```

```
uppercase_list_of_values = ABCABC
uppercase_list_of_values_without_lambda = ['A', 'B', 'C', 'ABC']
joined_list_of_values = abcabc
```

- the two lambdas are of course equivalent to named functions but avoid polluting the namespace and are easier to read as the are defined right where they are used
  - and cannot be used elsewhere so often reused functions should not normally be lambdas

```
def uppercase(x):
    return x.upper()
def joinstr(x,y):
    return x+y
```

## 1.5 Exercises

### 1.5.1 Random walkers

Write a code where two people perform a random walk along a rectangular `10x10` grid of coordinates, stopping when the hit (occupy same coordinates) each other for the first time.

## 1.6 Good Programming Practice: modularity

- a rule of thumb: *a single modular piece of code fits on screen all at the same time*
- split code into different files appropriately
  - in C/Fortran use a makefile to help compiling and linking them together
  - in python, codes in separate files become *modules*

## 1.7 Modules

- one has to *import* a module before it can be used
- python comes with a *standard library* of modules, see python standard library reference for details

- one such module is called `sys` and it know, e.g. your python version and more importantly, it holds the *module search path*: the list of directories python looks for `X` when it encounters a statement `import X` or `from X import Y`

```
import sys
print("Your python version is "+sys.version)
print("Your python module search path is "+",".join(sys.path))
```

```
Your python version is 3.4.2 (default, Oct  8 2014, 10:45:20)
[GCC 4.9.1]
Your python module search path is ,/home/juhaj/venv_teaching/lib/python3.4,/home/juhaj/
```

- the "current" directory is *always* searched first
  - "current" means the working directory for interactively started interpreter (i.e. without a script argument)
  - "current" means the directory containing the script being ran for non-interactive use
  - this can be confusing: Suppose `script.py` contains the statement `import z` and `z.py` is located in `/scriptdir` along with `script.py`. Then `cd /directory; python /scriptdir/script.py` will find `z`, but `cd /directory; python` followed by an interactive `import z` will fail yet `cd /scriptdir; python` followed by an interactive `import z` will again work:
  - for example we have a module called `MyModule.py` in `codes/python/MyModule.py` so let's import that

```
import os
print("Current working directory is "+os.getcwd())
import MyModule
print("The variable MyModule.module_internal_variable has the value "+str(MyModule.modu
```

```
Current working directory is /home/juhaj/IPCC/teaching/topics-python-in-research
>>> The variable MyModule.module_internal_variable has the value 42
```

- so that was the latter case of `z` above, now run it as a script

```
import subprocess
subprocess.Popen(["python", "codes/python/ImportMyModule.py"]).wait()
```

```
Current working directory is /home/juhaj/IPCC/teaching/topics-python-in-research
The variable MyModule.module_internal_variable has the value 42
```

- the search path is partially system dependent, but there's always `PYTHONPATH` which is searched before the system depedent path, so we can fix this

```
import os
import sys
print("Current working directory is "+os.getcwd())
sys.path = ["codes/python"] + sys.path
import MyModule
print("The variable MyModule.module_internal_variable has the value "+str(MyModule.modu
```

```
Current working directory is /home/juhaj/IPCC/teaching/topics-python-in-research
The variable MyModule.module_internal_variable has the value 42
```

## 1.8 Namespaces

- each variable lives in a *namespace*
  - like the above `MyModule.module_internal_variable` the part(s) before the dot specifies a namespace
- when you reference a variable, python searches for the name in several namespaces, starting from the most specific one:
  - the innermost scope (current module/source file, class, function)
  - the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope
  - the current module's global names
  - the outermost scope is the namespace containing built-in names

### 1.8.1 Good Programming Practice: it is a good idea not to "pollute" your namespace

- removes name clashes (relevant especially for short variable names like `temp` or `i`)
- protects from accidental modifications of wrong variables (e.g. due to typo etc)

  - note that python does not provide hard protection: there is **always** a way to alter the value of everything

- makes code easier to read, undersand, modify, and track what's happening to variables

- please do not `from x import *` it pollutes the enclosing namespace

### 1.9 Some standard modules

- we have already encoutered the `os` and `sys` modules: they are part of the python standard library

  - of particular interest might be `sys.stdin`, `sys.stdout`, and `sys.stderr`

`re` regular expression facilities, e.g.

```
import re
re.sub(r'(\b[a-z]+ )(\1)+', r'\1', 'please remove repeated repeated repeated words')
```

`urllib` we have already seen what this can do: access data using a URL

`datetime` everything you ever wanted to do with dates and timezone-less times between 0.0.0 CE and 31.12.9999 CE

- for proper timezone support, an external module called `pytz` is needed

`timeit` you may want to use `timeit.Timer()` instead of the next module for some performance measurements

`cProfile` performance profiler, we'll get to know this later

`doctest` a handy code quality checker which runs tests embedded into the docstrings: you will notice I already hid some of these in the earlier examples

```
import doctest

def daxpy(a, x, y):
    '''
    Calculate a*x + y.
    >>> daxpy(2.0,3.0,4.0)
    10.0
```

```
    '''
    return a*x+y


def daxpy_fails(a, x, y):
    '''
    Calculate a*x + y.
    >>> daxpy_fails(2.0,3.0,4.0)
    10.0
    '''
    return a*x+y+1

doctest.testmod()
```

```
**********************************************************************
File "__main__", line 4, in __main__.daxpy_fails
Failed example:
    daxpy_fails(2.0,3.0,4.0)
Expected:
    10.0
Got:
    11.0
**********************************************************************
1 items had failures:
   1 of   1 in __main__.daxpy_fails
***Test Failed*** 1 failures.
TestResults(failed=1, attempted=2)
```

`unittest` a more sophisticated testing environment; an even better one is available in an external module called `nose`

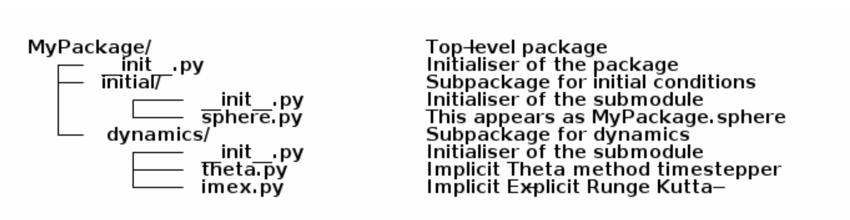
### 1.9.1 Packages: sets of modules organised in directories

- example

```
import scipy
import scipy.fftpack
print(scipy.fftpack.fftn([1,2,3]))
```

```
[ 6.0+0.j        -1.5+0.8660254j -1.5-0.8660254j]
```

- `scipy` is both the name of the package and its "main" module; this module contains a submodule called `fftpack`, which in turn contains a function `fftn` which simply calculates the FFT of the argument of arbitrary dimension
- such packages should be organised in directory trees




- **N.B.** when searching for a module to import `import MyPackage` will give priority to `MyPackage/__init__.py` over `MyPackage.py`
- `import MyPackage` will load and execute `MyPackage/__init__.py`: this is a magic file-name
- likewise `import MyPackage.initial` will execute `import MyPackage/initial/__init__.py`
  - note that `import MyPackage.initial` will implicitly also `import MyPackage`
  - some packages have convoluted directory structures and/or submodule handling which may prevent you from importing a submodule before explicitly importing the supermodule
- importing siblings must be done using the syntax `from MyPackage import Sibling`, e.g. `dynamics/__init__.py` imports `initial` with `from MyPackage import initial`
- the package `MyPackage` is available in the repo in `examples` directory

## 1.10 Exceptions

Whenever things go horribly wrong, python interpreter will *raise* and *exception*. If unhandled, these will cause the interpreter to exit with error, but not all errors are fatal, some can be handled. For this, python provides a `try-except-else-finally` construct. It is best described with an example

```
a="string"
b=10
try:
    c = a + b
except TypeError as arbitrary_variable_name:
    print("A TypeError was raised with the follwing arguments:")
    for i,arg in enumerate(arbitrary_variable_name.args):
        print("Argument #{i}: {a}".format(i=i, a=arg))
    c = str(a) + str(b)
except AttributeError as ae:
    print('Our example "c = a + b" can never raise this error.')
    print('But this is how you except many different types of exceptions if you need to
else:
    print("This will not execute as we ran into the except: -clause")
finally:
    print('This is executed as the very last thing of the construct. It is *always* exe
    print('As you can see, we have now set c to '+c)
try:
    c = a + a
except (TypeError, AttributeError, OSError) as arbitrary_variable_name:
    print("We do not come here, but this is how to handle multiple exception types in o
else:
    print("This will now execute as we did not run into the except: -clause")
finally:
    print('This is executed as the very last thing of the construct. It is *always* exe
    print('As you can see, we have now set c to '+c)
```

```
A TypeError was raised with the follwing arguments:
Argument #0: Can't convert 'int' object to str implicitly
This is executed as the very last thing of the construct. It is *always* executed.
As you can see, we have now set c to string10
This will now execute as we did not run into the except: -clause
This is executed as the very last thing of the construct. It is *always* executed.
As you can see, we have now set c to stringstring
```

- if you ever need access the attributes of "catch all" exception, you cannot use the `except X as Y` construct, but python provides the exception object via `sys.exc_info()` if necessary:

```
a="string"
```

```
b=10
import sys
try:
    c = a + b
except:
    exc = sys.exc_info()
    print("An Exception of type {} was raised with the following arguments:".format(exc
    for i,a in enumerate(exc[1].args):
        print("Argument #{i}: {a}".format(i=i, a=a))
    c = str(a) + str(b)
```

```
An Exception of type <class 'TypeError'> was raised with the following arguments:
Argument #0: Can't convert 'int' object to str implicitly
```

- the traceback is also available as the third element of the tuple: `exc[2]`

### 1.11 On I/O

- you should rarely, if ever, need to read a file using standard python routines
  - high performance (numerical) libraries are always more efficient for actual data
  - for "normal" small files, there is almost always a more high-level approach available either in standard python (like `sqlite3` module for sqlite databases or `email` for email messages etc)
  - sometimes `numpy` can be used to import even non-numerical data (`numpy.genfromtxt`)
- when you still need the low-level file operations, you should almost always use the `with` statement

```
import tempfile
placeholder_please_ignore_me=tempfile.NamedTemporaryFile()
filename = placeholder_please_ignore_me.name
print(filename)
with open(filename, "w") as f:
    f.write("this writes one line in the file\n")
    f.write("this writes part of a line ")
    f.write("this finishes the above line\n")
    f.writelines(["this writes\n", "all the\n", "list elements\n",
```

```
                  "in a sequence\n"])
print("The file object f is now closed: "+str(f))
rlen=10
with open(filename, "r") as f:
    some_data=f.read(rlen)
    lines=f.readlines()
print("The file object f is now closed: "+str(f))
print("The .read method read {l} bytes: {d}".format(l=rlen, d=some_data))
print("The .readlines method read from current file location to the end:\n"+"".join(
    lines))
placeholder_please_ignore_me.close()
```

```
/tmp/tmpmfq8m5f7
The file object f is now closed: <_io.TextIOWrapper name='/tmp/tmpmfq8m5f7' mode='w' en
The file object f is now closed: <_io.TextIOWrapper name='/tmp/tmpmfq8m5f7' mode='r' en
The .read method read 10 bytes: this write
The .readlines method read from current file location to the end:
s one line in the file
this writes part of a line this finishes the above line
this writes
all the
list elements
in a sequence
```

- the standard library provides a module `io` and class `io.StringIO` which for all practical purposes is a file, but only exists in memory
- the `mmap` module provides access to *memory mapped* files (also called *memmap*)
  - these look and feel like both strings and files (but are mutable unlike strings)
  - these are files which are accessed as if they were memory
  - they are only loaded into memory as needed, so you can memmap as big files as you wish without risk of running out of memory
  - but be careful, making copies etc of the data will **not** stay in the memmap file, so you may run out if you are not careful
- a couple of other useful I/O modules are `json` and `pickle`
  - `json` is the de facto standard data interchange format over internet and across architectures and programming languages

- `json` is not high performance or parallel, do not use with bigger than kB-range data
- `pickle` is python's `json` on steroids; in particular it can and will serialise python objects, but if you write client-server-type programs and pass data using `pickle` be mindful of the fact that untrusted clients can *send you arbitrary code* to be executed without explicit execution
  - not so with `json` unless you explicitly pass data from `json` to be executed

```
import pickle
import numpy
data=numpy.random.random(1000)
pickled=pickle.dumps(data)
print("Data size: {len} (plus small python object overhead)".format(len=data.nbytes))
print("Pickled size: {len}".format(len=len(pickled)))
unpickled=pickle.loads(pickled)
print("Note the type: "+str(type(unpickled)))
```

```
Data size: 8000 (plus small python object overhead)
Pickled size: 8159
Note the type: <class 'numpy.ndarray'>
```

## 1.12 Classes

### 1.12.1 Terminology

**class** defines a type of object, kind of glorified struct or you can think of birds

**instance** a representative of a class, think of birds again

**inheritance** classes form an "ancestry" tree, where "children" inherit "parents", but this is a very liberal family so a child can have an arbitrary number of parents (including 0 in python v2, but in v3 all children implicitly inherit "object")

**method** basically a function defined inside the namespace of a class

**attribute** a variable defined on the class namespace is a *class attribute*, be careful: only use immutables here; a variable defined inside a class method is an *instance attribute* and gets attached to the instance (like the `self.flies` below)

```
class animalia(object):
    '''animalia has two class attributes: level and heterotroph; they can be
       accessed by "self.level" and "self.heterotroph" inside the class and
       by "instancevariablename.level" and "instancevariablename.heterotroph"
       just like instance variables.
    '''
    level = "kingdom"
    heterotroph = True
class plantae(object):
    level = "kingdom"
    autotroph = True
class chordata(animalia):
    level = "phylum"
    notochord = True
class dinosauria(chordata):
    level = "clade"
    legs = 4
    def eat(self, food):
        '''Instance method which outputs a description of how dinosaurs eat.
        The first parameter is by convention called self, but there is no
        restriction on its name.'''
        print("Eating {f} with a mouth.".format(f=str(food)))
class tyrannoraptora(dinosauria):
    level = "clade"
    hollow_tail = True
class aves(tyrannoraptora):
    level = "taxonomical class"
    heart_chambers = 4
    def __init__(self, flight):
        '''When instantiating an aves we want to define whether it is capable
        of flight or not and save this information in an instance attribute
        "flight", note that instance attributes always need to be prefixed by
        "self." or whatever the name of the first parameter of the method is.
        Unprefixed variables become method local and cannot be seen from the
        outside.
        '''
        self.flight = flight
magpie=aves(True)
print("A {name} is an instance of {klass}.".format(name="magpie",
                                                    klass=magpie.__class__.__name__))
```

```
inheritancelist = magpie.__class__.mro()
for idx,_class in enumerate(inheritancelist[:-1]):
    print("The class {child} derives from {parent}.".format(
        child=_class.__name__, parent=inheritancelist[idx+1].__name__))
```

```
A magpie is an instance of aves.
The class aves derives from tyrannoraptora.
The class tyrannoraptora derives from dinosauria.
The class dinosauria derives from chordata.
The class chordata derives from animalia.
The class animalia derives from object.
```

### 1.12.2 Decorators and higher order functions

- A higher order function is a function which returns a function, like

```
def hello():
    print('''I'm a lowly function, returning an non-function object.''')
    return None

def HigherOrder(param):
    print('''I'm a higher order function: I return a function object.''')
    return param

this_is_a_function=HigherOrder(hello)
print("See what got printed!")
this_will_be_None=this_is_a_function()
```

```
I'm a higher order function: I return a function object.
See what got printed!
I'm a lowly function, returning an non-function object.
```

- note that the output of `hello` only appears at the very end: neither `def` nor `HigherOrder(hello)` causes the function body of `hello` to be executed

- N.B. `this_is_a_function` is the function `hello` *at the time the call to* `HigherOrder`: later redefinition of `hello` does not change `this_is_a_function`

```
def hello():
    return 42
```

```
newres=hello()
oldres=this_is_a_function()
print(newres==oldres)
```

```
I’m a lowly function, returning an non-function object.
False
```

- a better version of this is to use a *closure*: a function object that remembers values in enclosing scopes regardless of whether those scopes are still present in memory
  - I’m lying of course: they are still in memory but there is no other way to access them

```
def HigherOrder(param):
    print('''I'm a higher order function: I return a function object.''')
    x = 42
    def hello2():
        print('''The value of param is {} but x = {}'''.format(param, x))
        return None
    return hello2

this_is_a_closure=HigherOrder(24)
print("See what got printed!")
this_will_be_None=this_is_a_closure()
print('Note how "hello2" does not even exist now (in this scope):', hello2)
```

```
I'm a higher order function: I return a function object.
See what got printed!
The value of param is 24 but x = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'hello2' is not defined
```

- perhaps the most usual use of higher order functions is to create a *decorator*
- a typical example is to have "read-only" class attributes
  - N.B. these are not really read-only, python does not have such a concept, but you will have to go through some loops and hoops in order to write to them so it protects from bugs.

- there is a built-in higher order function for this, `property`, and also a short-hand syntax for wrapping functions in higher-order functions
- the following is a rewrite of the previous example with `HigherOrder`: note how `hello` itself now takes the place of `this_is_a_function` and that "pure" `hello` no longer exists: it is always wrapped in `HigherOrder`

```
def HigherOrder(param):
    print('''I'm a higher order function: I return a function object.''')
    return param

@HigherOrder
def hello():
    print('''I'm a lowly function, returning an non-function object.''')
    return None
```

```
... ... >>> ... ... ... ... I'm a higher order function: I return a function object.
```

```
this_will_be_None=hello()
```

```
I'm a lowly function, returning an non-function object.
```

- this is how `property` is usually used

```
class MyClass(object):
    def __init__(self, val):
        self._prop = val
    @property
    def prop(self):
        return self._prop

my_instance = MyClass(42)
print(my_instance.prop)
my_instance.prop = 0
```

```
42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

- `property` is actually a class, which eats functions in its `__init__`
- it also has methods `setter`, `getter`, `deleter` which can be used to allow setting and deleting such guarded attributes
  - `@property` is just a shorthand for specifying the `getter`
  - all of this is just shorthands: see `help(property)` for the longer syntax

```
class MyClass2(object):
    def __init__(self, val):
        self._prop = val
    @property
    def prop(self):
        return self._prop
    @prop.setter
    def prop(self,x):
        self._prop = x

my_instance = MyClass2(42)
print(my_instance.prop)
my_instance.prop = 0
print(my_instance.prop)
```

```
42
>>> 0
```

### 1.13 Little Bits

- `dir()` is a nice way to look at your namespace
  - can also be used to check if variable exists: `"variablename" in dir()`
- constructs like `for x in range(0,4)` can use an *iterator* or *generator* to produce the values for x
- have a look at python's docs for iterator and generator: we do not delve into them in this course but they are terribly useful

## 1.14 Exercises

### 1.14.1 More unit testing

- Implement more unit tests for some of the above function(s)

### 1.14.2 A tree using classes

- Create to a family tree for `magpie` which contains parent class instances

### 1.14.3 Fibonacci

How could one lecture programming without writing a Fibonacci code? By having students write it, of course, so please test your python skills by writing a program which computes the 100 first Fibonacci numbers and prints them out as a comma and space separated list, like `1, 1, 2, 3` etc splitting the output into lines of as close as possible but no more than 80 characters long.

Remember, this course is also about good programming practices, so make sure your Fibonacci-generator is reusable and has unit tests.

### 1.14.4 Game of Life

Write a simple `5x5` square game of life:

1. Any live square with fewer than two live neighbours dies, as if caused by under-population.
2. Any live square with two or three live neighbours lives on to the next generation.
3. Any live square with more than three live neighbours dies, as if by over-population.
4. Any dead square with exactly three live neighbours becomes a live square, as if by reproduction.
5. Boundaries are periodic to simulate infinite space (i.e. square at (xmax,y) has its "right" side neighbour at (0,y) etc).

You should write two functions: one to initialise the game and one to take a step. The initialiser should take the size of the game as an argument and return the state of the game. The stepper should take the current state as an input and return the new state.

We will later use these components to visualise the game, but for now please just print "X" and " " for the cells or just trust your skills and ignore output completely.