Ilkka’s Python Survival Kit

This document describes the elements of Python syntax/language that we cover in MARV216. This quick guide was written for my own learning purposes.

Important built-in functions

abs()	Absolute value |x|
bool()	Returns a bool, which can be True (1) or False (0)
chr()	For a number (0-255) , returns corresponding character
dict()	Returns a new dictionary. {key:value, key:value}
eval()	The *expression* argument is parsed and evaluated as a Python expression.
float()	Forces the argument to be a float, if possible
format()	Convert a *value* to a “formatted” representation, as controlled by *format_spec*.
globals()	Return a dictionary representing the current global symbol table
help()	Invoke the INTERACTIVE built-in help system.
id()	Tells where in the memory the variable is (starts from)
input()	Equivalent to eval(raw_input(prompt)).
int()	For converting numbers into integers (converting also between different number bases)
len()	Tells how many items there are in an object (sequence or mapping)
list()	Returns a list, arguments can be e.g. str, tuple. No argument -> empty list.
locals()	Returns a {} with current local symbol table.
long()	Convert to long integer.
max()	
min()	Returns maximum or minimum, of 2 or more values.
open()	Opening file objects for reading/writing to files. Returns a file object.
ord()	For a character (str of len 1), return the value of the byte (or two bytes for unicode str)
print	Print *object*(s) to the stream file, separate with commas.
range()	Gives a list of integers, from 0 to N-1, or from N to M, step is changeable
raw_input()	Returns a string, stripping the "\n" at the end (from pressing 'enter').
round()	Returns a float of first argument x rounded to n digits (second argument).
sorted()	Return a new sorted list from the items in *iterable.*
str()	Forces the argument to be a string, returns it.
tuple()	Returns a tuple.
type()	Tells the type of the argument (variable, constant, expression)
xrange()	Use this instead of range() in a for loop, if n in range(n) is really big (memory issues).

Modules

```
# Use help() to learn about functions and constants.
import 'module'
help ('module')
```

import math

ceil(), fabs(), factorial(), floor(),
exp(), log(), sqrt(), cos(), sin(),
acos(), asin(), .pi, .e etc.

import random

random()	Even distribution. x's between 0-1.
gauss(mu,sigma)	Gaussian, normal distribution, mean == mu, sdev = sigma.

import sys help(sys) - information about the implementation of Python on your machine.

There are many more in the standard library, and much more that can be installed on top of Python and be imported to your project (For example we will use numpy -package in the LiDAR project). Google "python built-in modules standard library".

Important keywords

Keywords must be spelled exactly as they are. Try `Import keyword ; keyword.kwlist`

and	(a and b)	(combines two booleans, 1 x 1 = 1, 1 x 0 = 0, 0 x 1 = 0, 0 x 0 = 0).
or	(a or b)	Returns also a boolean. 1+1 =1, 1+0 =1, 0+1=1, 0+0=0.
not	not(boolean)	Changes True to False and vice versa.

break	for i in range(10):
	  if a == b:
	    break
		(break out from a while/for loop in the midst)

class	class Tree:
	  def __init__(self, dbh):	

continue	for i in range(10):
	  if a == b:
	    continue
		(moves the control right back to the top of the while/for loop)

def	def MyFunc(x):
	  <body>
		(denotes a start of a function definition, is followed by identifier)

del	del indentifier	Deletes the reference to an identifier. If it is the only reference, memory is freed.
	del identifier[index]	A list item, and a dictionary item can be deleted (or a slice of a list).

if
elif
else
	if a == 1:
	  print "a is one"
	elif a == 2:
		Controlling program flow. if and elif are followed by expression that results in a boolean. If the value is True, the code (indentation) is executed.

```
  print "a is two"
else:
  print "a is not 1 or 2"
```

except
try

```
try:
  print "Attempting to divide with zero"
  a = 1/0
except:
  print "it failed"
  a = 1E200
```

Preparing for a potential error.
Try: <something prone to error>
Except <Do this is the try-thing leads to Error>

for starts a definite loop. Followed by "variable in sequence" structure.
while starts an indefinite loop. Followed by an expression resulting in boolean.

pass Null operation. Decorative keyword. Does nothing.

import
from import modules. use **"from math import sin"**, to get just single elements. sin() works then.

in 5 in range(5) results in True. Checks if something is in a data structure.
Works with tuples, lists, and dictionaries. ("Kalle") in {"Kalle":1, "Jussi":2} results in True.

(is) Compares if two identifiers actually are the same (point to the same storage address, id())

(lambda) Creates a small anonymous function.

print print a, b, c, d

return return back from a function/method. **return var1, var2,...**

N/B
None is an identifier.

Operators

=	assignment	
+= -= *= /=	incremental assigments: var = var @ something	
******	power numeric**numeric	
*****	int*float, int*int (multiplication),	str*int, list*int (copying the sequence)
+	int+float, int+int (addition),	str+str, list+list (concatenation)
-	int-float, int-int (subtraction)	
:	Slicing of, lists, strings, tuples (things/sets with an order) ":" starts also a body of a function, for, while, if, elif, else code block.	
%	modulo. % is sed also in string formatting.	
==	comparisons for logical expressions	
!= or <>	comparisons	
> < >= <=	comparisons	

Conditional – program flow

```
if (logical expression):
       <do this and break>
elif: (another logical expression):
       <do this and break>
elif: (another logical expression):
       <do this and break>
else: (we end up here only if the previous expressions yield False)
       <do this>
This is where control comes from "a break"
```

```
try:
       <Try something that may cause an error, if it doesn’t, break>
except:
       <Do this if an error was raised in the try-part>
```

Loops - program flow

```
for variable in sequence:
        <body of loop using variable starts>
        clever code
        if something:
                break   (leave the for loop at once, don't execute len(sequence) times)
                continue (leave the rest of the loop code untouched, and go back to the for -line)
        if something_else:
                This is not executed if continue keyword was executed earlier.
        <body of loop using variable ends>

NextLineOfCodeAfterAForLoop
<Control comes here after the loop is fully executed, or if break statement was execute>

while (something is true):
        <body of Loop starts>
        clever code
        if something:
                break
                continue
        <body of Loop ends>
```

In a while-loop the behavior of break and continue are the same as in a for-loop.

Data collections

list [], indexed items append(object), count(value), extend(iterable), index(), insert(), pop(), remove(), reverse(), sort(),...

For storing data records, when there is no unique (meaningful) key available.
mutable, easily changed, can store anything. Indices start from 0.
MyList = [1,2,3]
MyList[0] = math.sin

tuple () indexed, immutable .count() , .index() –methods. Slicing works. A tuple is list pruned from methods. Tuples are good for holding static data records.

dict {} key: value -pairs Unordered, slicing won't work. Access to an item through key. a[key]. Many handy methods: e.g. has_key(), items(), keys(), values(), del, pop(), popitem() etc.

Classes

```
class Tree1:
  def __init__(self):
    self.species = None
    self.dbh = None
    self.h = None
  def volume(self):
    return 0.5*math.pi/4*self.dbh**2*self.h

class Tree2:
  def __init__(self, species, dbh, h):
    self.species = species
    self.dbh = dbh
    self.h = h
  def volume(self):
    return 0.5*math.pi/4*self.dbh**2*self.h

Creating an instance of a Tree
MyTree = Tree1()
MyTree = Tree2('pine',0.321,19.2)

#---------------------------------------------------------------------------
# Creating 5 instances of a tree and adding them to a forest. Tree class is
# added Tree's 2D position as a new class.

import random, math

class Point2D:
  def __init__(self, x,y):
    self.x = x
    self.y = y

class Tree:
  def __init__(self,  species, pos, dbh, h):
    self.species = species
    self.pos = pos
    self.dbh = dbh
    self.h = h
  def volume(self):
    return 0.5*math.pi/4*self.dbh**2*self.h
```

```
class Forest:
  def __init__(self, area):
    self.area = area
    self.trees = []

# ---------------------------------------------
MyForest = Forest(0.5)

for i in range(5):
   # Draw values for the attributes
   dbh = 0.3 + random.random()/10.0
   h = 20 + random.random()* 2.0
   x = random.random()* 20.0
   y = random.random()* 20.0
   Pos = Point2D(x,y)
   MyForest.trees.append( Tree('pine', Pos, dbh, h ) )

for i in MyForest.trees:
  print i.pos.x, i.pos.y, i.species, i.dbh, i.h, i.volume()
# -----------------------------------
```

<u>Reading the code</u>

a[b](c)

- "**(c)**" indicates that the object left of it has to be an identifier of a method or a function. **c** is an argument.
- "**[b]**" is clearly saying that **a** is an identifier to a sequence that can be indexed and **b** is the index. So **a** can be a dict, tuple, list, or a string. Except that a string cannot hold a function reference, so **a** has to be dict, tuple,or a list.
- **a[b]** is the thing left of **(c)**, so it hold the method/function.
- Actually, when you think of it, **a[b]** could also be a class (constructor), as they are callable (like functions).

a.b
a.c()

The dot notation says that **a** has to be an object. Objects can have attributes and methods. a.b refers to attribute b in a, while a.c() calls the method c in a.

a[2:]

This tells that slicing is meaningful, so **a** has to be a str, tuple, or list.

a[0] = b

This tells that assignment is ok, **a** has to be a list or a dict.

for x,y in a:

This tells that entries in **a** are tuples/lists with two items/objects in each.

for x in a:

This tells that entries in **a** are single objects. They can be dictionaries even.

x in a:

This tells that a is a sequence.

a.b.c

This tells that attribute b in object a is a class instance, with attribute c

a.b[0]

This tells that attribute b in object a is a data collection or a string.

if a

This tells that a has to be a boolean

a.upper()

This tells that a is a str.

(a.f(',')).sort()

This tells that a.f() must return a list, for sort() to make sense.

a % b

This tells that a and b have to be numeric variables(int/float).

print len(a), a

This would fail.

def __init__(self):

An object from this class is initiated without supplying any arguments for attributes.

a = random.random()

a will be a float, and take values from 0.0 to 1.0.

File I/O

File objects are central: assign with open() function: **filevar= open("c:\\temp\\aurora.py","r")**
Modes are "r", "w", "a" for reading, writing and appending.
filevar.close() close the file (tells the operating system we don't need it)

Reading is easy using file object's methods

.read()	The whole contents in a string (with \n's). Call just once.
.readline()	Returns the next line. (with \n's)
.readlines()	Returns a list with lines as entries.

If we iterate thru the file object, we get lines.

```
filevar= open("c:\\temp\\aurora.py","r")
for line in filevar:
    print line
```

Writing requires that the file is opened in either "a" or "w" mode. "w" kills the file if it exists. "a" will take the write-head to the end of the file, when writing starts. .write() method is used for writing strings (text).
filevar= open("c:\\temp\\aurora.py","w")
filevar.write("String\n")

String formatting/processing

With str**.format()** method, or with the **%** (string formatting operator), or format() built-in.

```
count = 1
print "This is count now %d" %count
print "This is count now %.2f" %count
print "This is count now %.10d" %count
print "This is count now %10d" %count
print "Tab\tTab\tTab\tTab"
print "{:>30}".format("Tab\tTab\tTab\tTab")
print "Newline\nNewline\n"
print "{:*^30}".format(3.1415)
print "{:.>+30}".format(3.1415)
```

```
This is count now 1
This is count now 1.00
This is count now 0000000001
This is count now          1
Tab   Tab   Tab   Tab
               Tab      Tab   Tab   Tab
Newline
Newline

************3.1415************
.......................+3.1415
```

with format() function.

```
>>> format(1.35,".3f")
'1.350'

>>> format(1.35,"10.3f")
'     1.350'

>>> format(1.35,"<10.3f")
'1.350     '

>>> format(1.35,".^10.3f")
'..1.350...'

>>>format("python",".>20s")
'..............python'

>>>format(135,"10d")
'       135'
```