An introduction to Python for Engineering (one day)

Rev #7

Internal Course for Volvo Cars R&D

Olof Attemo, CAE Support Team, osvens13@volvocars.com

Andreas Hall, CAE support team, ahall169@volvocars.com

David Andersson, CAE Support Team, dander12@volvocars.com

Motivation for this course

This course aims to quickly teach you basic concepts and get you started and oriented in the python language and start applying it to science and engineering. After the course you should be able to read, understand and write python code to load, manipulate and plot data, navigate the python library documentation and continue to learn more about the language and its many applications on your own.

Part I – The python language

Why python is so popular

- It is easy to apply python to existing APIs written in C,C++,Java and C#
- A lot of modules exist for various tasks from engineering to web development
- Many CAE-Applications can be controlled by Python-scripts: Ansa, Abaqus, Adams etc.
- Designed to be highly readable and easy to write
- “Interpreted” language - No compiling required
- Read more at www.python.org

Running the interpreter, interactive and non-interactive modes

From the start-menu, go to the Anaconda3 (python 3.x) folder, and start “Spyder”. The Spyder IDE (Integrated Development Environment) features a code editor and an *interactive python shell*.

In the shell you can type python commands and they will execute immediately in the environment that the new shell defines. The concept is similar to MATLAB and other scripting languages.

```
>>> print('Hello')

Hello
```

A python code file is called a module. In most cases, a module is saved to a .py file and run in *non-interactive mode*, meaning without user interaction.

Select File > New File

The editor comes up.

Copy and paste the command you just typed, from the shell to the editor.

Save the file to for example $HOME\python\hello.py

Select Run > Run

```
Hello
```

Another way of running your scripts in non-interactive mode is by passing them as a command-line argument to the python interpreter:

```
$HOME\python>python hello.py
```

```
Hello
```

Basic syntax and types

Python is inherently *object-oriented*, in the sense that functions and variables (almost) always belong to an object. An object is a named entity that has a type. The type is the object identity and defines how the object behaves, what it contains and what operations can be performed on it. Everything in python except some built functions conforms to this model.

All variables are represented as objects. Python has a number of built-in types such as scalars, tuples, strings and lists. Custom types may also be defined. In object-oriented programming, the definition of a user-defined object is commonly referred to as a ‘class’. In their most basic form, these are composed using a combination of built-in types and user-defined functions. Class objects are at the heart of many python interfaces. For basic python scripts it is often not necessary to define your own classes, but it is important to distinguish them from the built-in types.

Strings

By using quotes in assignment, python knows that we want to create a string object. Each built-in type has its own special assignment notation.

```
>>> a = 'Hello'
>>> a
'Hello'
```

The variable "a" now represents a string object, which has the value "Hello". Objects generally contain *members*. Members can be of any type, but usually we want to access *member functions* to perform some operation on the object.

To access a member function we use “dot notation”. The string type has the member function “lower”, which in our case returns a new string object with the value “hello”.

```
>>> a.lower()
'hello'
```

In python strings are represented in the current console character set. This has portability implications when using regional non ASCII-characters, such as the Swedish vowels å, ä and ö. In that case we might want to explicitly use Unicode strings instead to always get the proper output.

Unicode strings (python 2.x)

```
>>> a = u'Örjan Gärdegås'
>>> a
u'\xd6rjan G\xe4rdeg\xe5s'
print(a)
Örjan Gärdegås
```

Print is a special built-in function which needs no parentheses to invoke. Most built-in functions however behave just like member functions, but can be reached anywhere in the code and have no parent object to which they belong. In python3 all strings are unicode strings, unless they are prefixed by the r character. For example: `r'a string` This means “raw string”. It could be in any format besides Unicode (typically ASCII). A raw string may be explicitly converted to Unicode.

Booleans

```
>>> a=True
>>> a
True

>>> not a
False

>>> a or True
True

>>> a and False
False
```

Integers

```
>>> a = 1
>>> a
1
```

Arithmetic on integers

```
>>> a = 2
>>> a + 1
3
>>> a - 1
1
>>> 2*a
4
>>> a/2
1
>>> a**2
```

```
4
>>> -a
-2
>>> abs(-a)
2
```

Floating point numbers

```
>>> a = 1.2
>>> a
1.2
```

Converting integers to floating-point numbers

```
>>> a = 2
>>> a= float(a)
>>> a
2.0
```

Lists

The built-in list type is the main concept for working with sequences of data in python. There is no inherent concept of numerical vectors or matrices like in MATLAB, but there are third-party libraries that provide such functionality. There is also a standard library module that can be used to represent *arrays* which are often used for larger datasets.

A list is a sequence of items of any type. The main differences between an array and a list are that elements can be added and removed without recreating the list, and that each element in the list is independent and may be of any type.

```
>>> a = [1,2]
>>> a
[1, 2]
```

The built-in function “range” can be used to generate a list of numbers 0 though 9

```
>>> a = range(10) # --> Python 3.x, Lazy object
>>> a = list(a) # --> Python 3.x, convert to list
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Determining the length of a list:

```
>>> len(a)
10
```

Accessing items in lists:

```
>>> a[0]
0

>>> a[10]
```

```
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    a[10]
IndexError: list index out of range
```

This an example of a *trace* which occurs when we provide invalid input. This particular trace lets us know that the first element in a list of length 10 is at index 0, and the last is at index 9.

```
>>> a[5]
5
>>> a[-1]
9
```

Selecting a range of items from a list:

```
>>> a[1:4]
[1, 2, 3]
```

Observe that this notation gets us a half-open interval where the fist index is included and the last index is not. We get element number one though (and not including) number four.

```
>>> a[1:]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[:1]
[0]
```

Adding items to a list:

```
>>> a.append(10)
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

When performing operations on lists, the member functions will generally change the items in the list "a" instead of returning a new object like we observed with strings. A list object is an example of what we call a *mutable* object. The list object represented by “a” is changed permanently. This is generally the case for complex objects like lists or class objects.

All simple objects, like scalars and strings, are only created once. These are called *immutable* objects. When an operation is performed on such an object, the result is created as a new object and the old object is thrown away. This is not immediately obvious but it has some important implications as we will observe later.

List items can also be lists, or any other type:

```
>>> a.append([11,12])
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, [11, 12]]
```

The number of items in this list is thus:

```
>>> len(a)
12
```

Removing items from a list:

```
>>> a.pop()
[11, 12]
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In the IDLE shell, you may type a. ("a dot") and then press TAB to get a list of methods that can be applied to this object.

Removing the first item of a kind if it exists:

```
>>> a.remove(10)
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Reverse the order of items:

```
>>> a.reverse()
>>> a
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

If you want to retain the original list "a" in a case like this, you might want to create a copy and store it as "b" for example.

```
>>> b = a
```

Assignment in this way will not produce the results we want. For mutable objects such as lists, assigning "a" to a new variable does not imply that we have created a new identical and independent copy of the object.

Sort the items in a list again. Easy for a sequence of integers

```
>>> a.sort()
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Now lets examine the list b:

```
>>> b
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The observation that "b" was also changed is very important. Variables in python that represent mutable objects do so by reference, and not by value. The list that "a" refers to is thus not copied to a new location referenced by "b", but the variable "b" will reference to the same object as "a". To create an independent copy of a list, we can use the built-in list constructor function `list()`:

```
>>> b = list(a)
>>> b.reverse()
>>> b
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Find the smallest number:

```
>>> min(a)
0
```

Find the greatest number:

```
>>> max(a)
9
```

Sum the items in the list:

```
>>> sum(a)
45
```

Now we know how to compute the average:

```
>>> sum(a) / len(a)
4
```

Whats wrong with this result? Nothing, since we are performing arithmetic on integers, not natural numbers as represented by double precision floating point as we might have intended. In integer division, the remainder is lost, but can be obtained using the modulus operator.

```
>>> sum(a) % len(a)
5
```

Tuples (immutable sequences)

```
>>> a = (1,2)
>>> a
(1, 2)
```

Comma-seperated numbers are interpreted as a tuple

```
>>> a = 1,2
>>> a
(1, 2)
```

Dictionaries

The dictionary type in python is used to store keys and values. The dictionary or objects with a *dictionary interface* for key-value storage is commonly used in python modules. Some database interfaces backed by files or more complex storage methods use a dictionary interface. Certain modules use dictionaries for storing settings.

```
>>> mydict = {'name': 'David', 'age': 36, 'height': 181}
>>> mydict['name']
David

>>> mydict['age']
36

>>> mydict['height']
181

>>> mydict['city'] = "Gothenburg"
>>> mydict['city']
Gothenburg
```

Loops

For-loops:

```
>>> a = range(10)
>>> for i in a:
          print(i)

0
1
2
3
4
5
6
7
8
9
```

While loops:

```
>>> while (len(a) > 5):
          print(a.pop())

9
8
7
6
5
```

```
>>> a
[0, 1, 2, 3, 4]
```

Conditionals

When defining conditionals or functions, each branch of code must have its own indentation level. Python enforces this. Failing to indent properly is a syntax error. We will be using tab for indentation. You can use spaces or tab for indentation as long as the indentation level is used consistently through each branch.

```
>>> if len(a) > 5:
          print('yes')
else:
          print('no')

"no"
```

Comparison operators

```
> greater than
< less than
== equal
!= not equal
>= greater or equal
<= less or equal
```

Defining a function

```
>>> def double(x):
          return 2*x
```

Keep in mind that python does not enforce types for function arguments. The “double” function does what you might expect for scalars:

```
>>> print(double(2))
4
```

But maybe not what you expect for lists (or any sequence like a tupe):

```
>>> print(double([1,2,3]))
[1, 2, 3, 1, 2, 3]
```

Operating on list elements

In this case you might want to 'map' the double function to your list

```
>>> print(map(double, [1,2,3]))
[2, 4, 6]
```

This operation can also be expressed as a so-called list-comprehension

```
>>> [double(x) for x in [1,2,3]]
[2, 4, 6]
```

Now we can improve our average-value calculation by converting our list of integers to a list of floating-point numbers using a list comprehension.

```
>>> a = range(10)
>>> [float(x) for x in a]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

Or by using map to apply the float function to each element in the list

```
>>> a = map(float, a)
>>> a
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

```
>>> sum(a) / len(a)
4.5
```

Since we are now using floating point numbers, we get the result we want.

The built-in and standard library modules

Modules, classes, objects and members

The module from where your python scripts are started is referred to as the main module. This is the module we are currently working with.

Let us have a look at what we have in our module. We can list all objects that are currently loaded using the built-in function `dir()`

```
>>> dir()
```

```
['__builtins__', '__doc__', '__name__', '__package__', 'a', 'b',
'double', 'x']
```

The variables begin and end with double underscore follow a widely used naming standard for private members of a module. An example is the '__name__' variable which contains the name of the current module as as a string. These variables are used internally by modules and classes. They can, but generally should not be used directly.

The function we defined previously, double, exists as an object in the module. The object type is function.

```
>>> double
<function double at 0x036E6CF8>
```

The output you get when referring to an object in the console is always the objects *string representation*. How an object is represented in string form depends on its type. All objects have a string representation. The default representation looks like the one above and lists the object type, the name and the address of the object in memory.

Python has a number of modules which define the standard APIs that come with python. They can be imported using the command "import".

Let's import the os module, which provides us with a number of basic functions from the underlying operating system.

```
>>> import os
>>> os

<module 'os' from 'C:\Python27\lib\os.pyc'>
```

The 'os' module has now been loaded into the main module. The os module is stored as os.py file in the python standard library directories. The .pyc file is a "compiled python" file which gets generated on the fly to to speed up the python interpreter. Compiled python is not machine code and thus not dependent on the machine architecture.

An imported module is loaded as an object of module type in the current module. The variables and functions in the imported module may then be accessed by prefixing their names with 'os.'

Two examples from the os module are the os.sep variable that contains the directory separation character for the current system, and the os.getcwd() function that returns a string with the current working directory of the python interpreter.

```
>>> print(os.sep)
\

>>> print(os.getcwd())
Y:\Data\python
```

The os module may in turn import other modules, which may import yet other modules. This will result in a tree-like structure of objects that define the name spaces for all objects currently available to our current module.

The python standard library

The python standard library contains a lot of functionality. The full list for the current version is available at the python website at: http://docs.python.org/library/index.html

Find chapter 9 in the library and go to the top level of 9.2, the math module, in the list and open it.

The documentation details the functions and constants available in this module. Note that there are no types defined for any arguments or any return values. Sometimes there are constraints defined for the input, sometimes not. In most cases it is implicit which type should be used and what type is returned.

This makes python very flexible, as the same function may operate on different input types, but it also forces you to keep track of what types your variables have. There is no early warning when you make a mistake. Instead the interpreter will stop with an error and a trace of the current call stack, to let you try to figure out what went wrong.

Scientific python (Numpy, scipy, matplotlib etc)

Many extension modules are not a part of the standard library. For science and engineering the most notable are numpy, scipy and matplotlib. These provide means of handling more sizable data sets efficiently, perform scientific operations and plot data. The functionality of these packages combined is similar to MATLAB with a basic set of toolboxes for statistics and optimization.

Exercise 1: Writing a simple script

In the examples folder there is a file called 'average.py', open this file in the editor. This module loads a list of numbers from the comma-separated vector file 'ratings.csv'. Choose File -> Open in the editor and load the csv-file as well. Have a look at the format of this file, but leave it as is. Then close it again.

Go to the average.py module windows and from the menu, choose Run and Run Module or press F5.

The numbers from the csv-file are printed to the python console. The objective with this exercise is to modify the script to calculate the sum and the average of these values, and print the result.

Exercise 1: Solution walk-through

The lab introduces some new concepts:

1. File objects
2. Classes
3. Optional arguments to functions
4. The 'reader' pattern (an object that is an *iterator* that reads lines)

A file object is simply a reference to an open file. The open function is built-in to the languge. The file object has several methods such as read, write and close.

The csv.reader is a class. A class is created as an object of the specific class type. The object works similar to a module, except that it is designed to be self-contained, and have any number of objects of its type created. In object oriented programming, member functions of a class object are called “methods”. A class often defines a special method called a constructor to initiate an object that is created from it. The constructor has arguments just like any other method. When the csv reader is created, its constructor is called in the way observed in the lab example.

Note that the last two arguments to the constructor are assigned values as the function is called. These arguments are optional and do not need to be specified at all. A function may have any number of optional arguments but they must appear after any ordinary (required) arguments.

As we iterate over a file object, or a csv reader object, it has the special behavior of reading the the file line by line. *An object with an interator interface behaves like a list*.

Sorting, searching and parsing

Sorting a list of lists

In a more complex scenario, such as when we have a list of iterables, it may become necessary to pass a key function as an argument to sort(). The key function helps identify which part of each element is relevant to look at for the sorting operation.

```
>>> def getfirstelem(x):
         return x[0]
```

```
>>> a = range(10)
>>> a = list(a)  # --> See page 6.
>>> a.reverse()
```

The zip function joins one or several lists like a zipper, and returns a list of tuples, each with one element from each list. The length of the resulting list is equal to the length of the shortest list in the argument.

```
>>> a = zip(a, range(10))
>>> a = list(a)  # --> See page 6.
>>> a
```

```
[(9, 0), (8, 1), (7, 2), (6, 3), (5, 4), (4, 5), (3, 6), (2, 7), (1,
8), (0, 9)]
```

```
>>> a.sort(key=getfirstelem)
>>> a
```

```
[(0, 9), (1, 8), (2, 7), (3, 6), (4, 5), (5, 4), (6, 3), (7, 2), (8,
1), (9, 0)]
```

Printing and formatting numbers

Notice that by default the python console prints full IEEE-754 “double precision” floating point numbers. That is why the results are not exact, but as they are rounded to 53 bits of binary fraction, they contain rounding errors.

This is the case in all applications that use double precision floating point as supported by common hardware. It is common in many languages to hide this by only printing a certain number of decimal places by default.

This also holds true for the built-in print function in python, when we explicitly print the floating point numbers:

```
>>> for i in [0.1*x for x in range(10)]:
          print(i)

0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
```

We can control this with formatting options when printing floating point numbers. Anyone who as had experience with C will recognize this.

```
>>> a = 0.2211111
>>> print('%.2f' % a)
0.22
```

Multiple arguments to print can be specified as a tuple

```
>>> b = a + 0.1
>>> print('%.6f %.4f' % (a, b))
```

Most objects have a string representation, which is what we get in the console if we access them by their names.

```
>>> print('%s' % range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print('%s' % len)
<built-in function len>
```

Search for a string in a string

```
>>> a = 'abcdefgh'
>>> a.find('ab')
0

>>> a.find('e')
4

>>> a.find('i')
-1
```

Split a string in tokens

```
>>> a = 'the quick brown fox jumps over the lazy dog'
>>> b = a.split(' ')
>>> b
['the', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy',
'dog']
>>> b[4]
'jumps'
```

To assemble a string again, we can use the join method of the string type. It is the string that contains the desired token separator that this function should be called on. We use a string containing a single space for this:

```
>>> ' '.join(b[:5])
'the quick brown fox jumps'
```

String parsing using regular expressions

In the python standard library there is a regular expression module, “re”, for advanced parsing of strings.

```
>>> import re
```

The re.search() method takes a regular expression pattern and a string and searches for that pattern within the string. If the search is successful, search() returns a match object or None otherwise. Therefore, the search is usually immediately followed by an if-statement to test if the search succeeded, as shown in the following example which searches for the pattern 'word:' followed by a 3 letter word (details below):

```
mystr = 'an example word:cat!!'
match = re.search('word:\w\w\w', mystr)
if match:
   print('found', match.group())
else:
   print('did not find a match')
```

The code `match = re.search(pat, str)` stores the search result in a variable named "match". Then the if-statement tests the match -- if true the search succeeded and match.group() is the matching text (e.g. 'word:cat'). Otherwise if the match is false (None to be more specific), then the search did not succeed, and there is no matching text.

Basic Regex Patterns

The power of regular expressions is that they can specify patterns, not just fixed characters. Here are the most basic patterns which match single chars:

a, X, 9, < -- ordinary characters just match themselves exactly. The meta-characters which do not match themselves because they have special meanings are: . ^ $ * + ? { [ ] \ | ( ) (details below)

. (a period) -- matches any single character except newline '\n'

\w -- (lowercase w) matches a "word" character: a letter or digit or underbar [a-zA-Z0-9_]. Note that although "word" is the mnemonic for this, it only matches a single word char, not a whole word. \W (upper case W) matches any non-word character.

\b -- boundary between word and non-word

\s -- (lowercase s) matches a single whitespace character -- space, newline, return, tab, form [\n\r\t\f]. \S (upper case S) matches any non-whitespace character.

\t, \n, \r -- tab, newline, return

\d -- decimal digit [0-9] (some older regex utilities do not support but \d, but they all support \w and \s)

^ = start, $ = end -- match the start or end of the string

\ -- inhibit the "specialness" of a character. So, for example, use \. to match a period or \\ to match a slash. If you are unsure if a character has special meaning, such as '@', you can put a slash in front of it, \@, to make sure it is treated just as a character.

Basic Regex Examples

Joke: what do you call a pig with three eyes? piiig!

The basic rules of regular expression search for a pattern within a string are:

- The search proceeds through the string from start to end, stopping at the first match found
- All of the pattern must be matched, but not all of the string
- If `match = re.search(pat, str)` is successful, match is not None and in particular match.group() is the matching text

```
  ## Search for pattern 'iii' in string 'piiig'.
  ## All of the pattern must match, but it may appear anywhere.
  ## On success, match.group() is matched text.
  match = re.search('iii', 'piiig') =>  found, match.group() ==
"iii"
  match = re.search('igs', 'piiig') =>  not found, match == None

  ## . = any char but \n
  match = re.search('..g', 'piiig') =>  found, match.group() ==
"iig"

  ## \d = digit char, \w = word char
  match = re.search('\d\d\d', 'p123g') =>  found, match.group() ==
"123"
```

```
  match = re.search('\w\w\w', '@@abcd!!') =>  found, match.group()
== "abc"
```

Things get more interesting when you use + and * to specify repetition in the pattern:

- + -- 1 or more occurrences of the pattern to its left, e.g. 'i+' = one or more i's
- * -- 0 or more occurrences of the pattern to its left
- ? -- match 0 or 1 occurrences of the pattern to its left

First the search finds the leftmost match for the pattern, and second it tries to use up as much of the string as possible -- i.e. + and * go as far as possible (the + and * are said to be "greedy").

Regex Repetition Examples

```
  ## i+ = one or more i's, as many as possible.
  match = re.search('pi+', 'piiig') =>  found, match.group() ==
"piii"

  ## Finds the first/leftmost solution, and within it drives the +
  ## as far as possible (aka 'leftmost and largest').
  ## In this example, note that it does not get to the second set of
i's.
  match = re.search('i+', 'piigiiii') =>  found, match.group() ==
"ii"

  ## \s* = zero or more whitespace chars
  ## Here look for 3 digits, possibly separated by whitespace.
  match = re.search('\d\s*\d\s*\d', 'xx1 2   3xx') =>  found,
match.group() == "1 2   3"
  match = re.search('\d\s*\d\s*\d', 'xx12  3xx') =>  found,
match.group() == "12  3"
  match = re.search('\d\s*\d\s*\d', 'xx123xx') =>  found,
match.group() == "123"

  ## ^ = matches the start of string, so this fails:
  match = re.search('^b\w+', 'foobar') =>  not found, match == None
  ## but without the ^ it succeeds:
  match = re.search('b\w+', 'foobar') =>  found, match.group() ==
"bar"
```

Regex Group Extraction

The "group" feature of a regular expression allows you to pick out parts of the matching text. Suppose for the emails problem that we want to extract the username and host separately. To do this, add parenthesis ( ) around the username and host in the pattern, like this: r'([\w.-]+)@([\w.-]+)'.

In this case, the parenthesis do not change what the pattern will match, instead they establish logical "groups" inside of the match text. On a successful search, match.group(1) is the match text corresponding to the 1st left parenthesis, and match.group(2) is the text corresponding to the 2nd left parenthesis. The plain match.group() is still the whole match text as usual.

```
  mystr = 'purple alice-b@google.com monkey dishwasher'
  match = re.search('([\w.-]+)@([\w.-]+)', mystr)
  if match:
    print(match.group())   ## 'alice-b@google.com' (the whole match)
    print(match.group(1))  ## 'alice-b' (the username, group 1)
    print(match.group(2))  ## 'google.com' (the host, group 2)
```

A common workflow with regular expressions is that you write a pattern for the thing you are looking for, adding parenthesis groups to extract the parts you want.

Regex findall

findall() is probably the single most powerful function in the re module. Above we used re.search() to find the first match for a pattern. findall() finds *all* the matches and returns them as a list of strings, with each string representing one match.

```
  ## Suppose we have a text with many email addresses
  mystr = 'purple alice@google.com, blah monkey bob@abc.com blah 
dishwasher'

  ## Here re.findall() returns a list of all the found email strings
  emails = re.findall('[\w\.-]+@[\w\.-]+', mystr) ## 
['alice@google.com', 'bob@abc.com']
  for email in emails:
    # do something with each found email string
    print(email)
```

For files, you may be in the habit of writing a loop to iterate over the lines of the file, and you could then call findall() on each line. Instead, let findall() do the iteration for you -- much better! Just feed the whole file text into findall() and let it return a list of all the matches in a single step (recall that f.read() returns the whole text of a file in a single string):

```
  # Open file
  f = open('test.txt', 'r')
  # Feed the file text into findall(); it returns a list of all the 
found strings
  strings = re.findall('some pattern', f.read())
```

Error handling

When an invalid expression is encountered, the default behavior of python is to stop execution and print the error. Often it is desirable to take action on the error and continue running. Consider the following:

```
>>> int('a')

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

ValueError: invalid literal for int() with base 10: 'a'
```

An error occurs because there is no obvious way of converting the 'a' character into an integer. Printed above is a traceback indicating where the error occurred, and an exception of the type "ValueError". An exception in the general sense has a type and a description provided by the code that throws the exception.

To define an error handler for the ValueError exception we define our own conversion function.

```
def string2int(s):
    try:
        return int(s)
    except ValueError:
        return None
```

The keyword try is used encapsulate the code that may cause an exception. The except-section allows for specification of an error handler for the ValueError type. If the exception triggered in the try section is not of a type specified among the handlers it will pass to the caller and result in a stop and traceback. The except keyword may be used without specifying a type to catch all exceptions. This is generally not desirable as it may mask issues that should be handled explicitly.

```
>>> print(string2int(1))

1

>>> print(string2int('a'))

None
```

Object oriented programming

A primitive class in python

A class in python is used to define a custom object with corresponding members/functions. Create a new file with the following contents and name it vehicle.py

```
class Vehicle:

    def __init__(self):
        print("Initializing..")
        self.vehicleType = 'Undefined'
        self.weight = 0

    def setType(self, newType):
        self.vehicleType = newType

    def setWeight(self, weight):
        self.weight = weight

    def getInfo(self):
        print('Vehicle')
        print('  |-->type: %s' % self.vehicleType)
        print('  |-->weight: %s'  % self.weight)

if __name__ == "__main__":
    v = Vehicle()
    v.getInfo()
```

The __name__ == "__main__": -if clause is a crude but standard way of telling the python interpreter to execute code only if the file is called to run directly, as an entry point for the application. Importing this file as a module will not trigger the code under this if-clause.

Run the file directly through Spyder by using the run command (F5). The result should come out as:

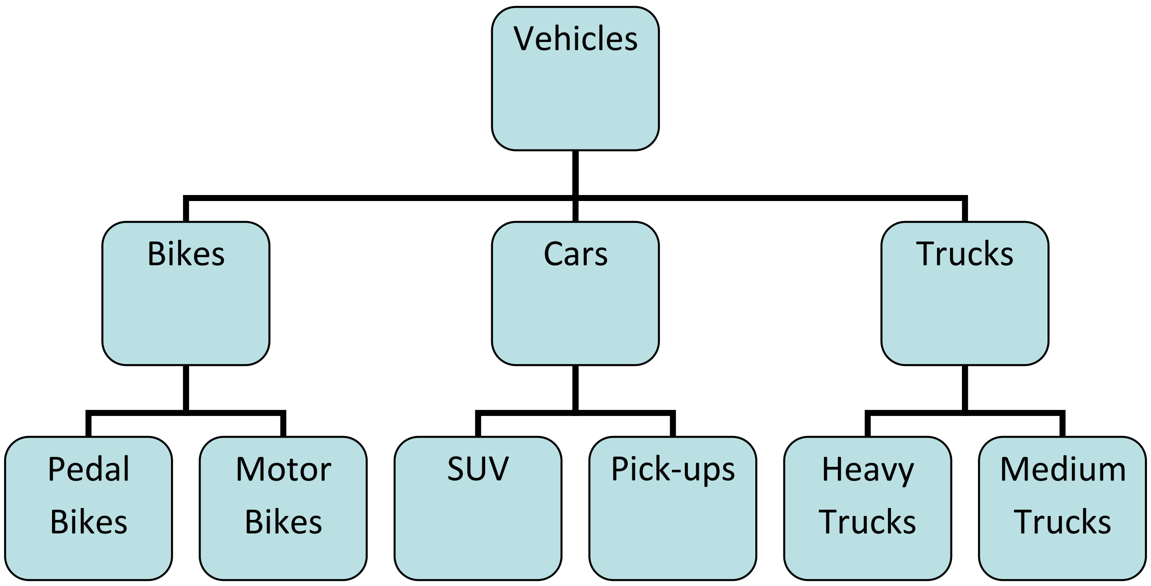
```
Initializing..
Vehicle
  |-->type: Undefined
  |-->weight: 0
```

In the above code, the class Vehicle is created as the object v. The object v is called an instance of the class. Like a variable, any number of instances of the class may be created. This is true for most classes. Notable exceptions include wrapper classes for hardware resources that have technical limits on concurrent access.

The function getInfo(self) is a referred to as a class member function or member function of the class Vehicle. The first argument of a member function is always "self". The "self" object is a reference to the current instance of the class object.

The function __init__(self) is called automatically when the object is created to allow for instance specific initialization before any member functions are called.

Object oriented design "OOD" is the process of structuring the classes of an object oriented implementation and their respective functions to. Goals usually include maximization of code reuse, easy maintainability and extendibility and logical abstraction and functional separation. This is achieved through features such as *inheritance* and *composition*.



Inheritance

Inheritance is a concept where a new class is derived from an existing class. The inherited class keeps all the functionality of the "parent class" and optionally overrides and adds members. Create a new file in the same directory as the vehicle.py file created previously but this time name it car.py, and enter the following:

```
from vehicle import Vehicle

class Car(Vehicle):

    def __init__(self, model='Undefined'):
        super().__init__()
        self.setType('Car')
        self.model = model

    def getModel(self):
        print('Model:', self.model)

    def setModel(self, model):
        self.model = model
    def getAllInfo(self):
        self.getInfo()
        print('  |__\n     |-->model:', self.model)

if __name__ == "__main__":
    v = Car()
    v.getInfo()
    v.getModel()
    print('-All info-')
    v.getAllInfo()
```

Run the car.py file in Spyder and observe the output:

```
Initializing..
Vehicle
  |-->type: Car
  |-->weight: 0
Model: Undefined
-All info-
Vehicle
  |-->type: Car
  |-->weight: 0
  |__
     |-->model: Undefined
```

Inspection reveals that the Car class contains the superset of class members from both the Vehicle (the parent class) and the Car class. The Car class inherits from the Vehicle class. Also notable is that the Car class defines its own __init()__ -method and calls the init method from the parent class using the built in super()-function take care of any Vehicle-specific initialization before initializing Car. In older versions of Python (<3.0), the super command looks like this:

```
super(Car, self).__init__() # Python 2.x style
```

The Car class also has one predefined value of the variable *model*. This is just to show how to define variable members and attributes to an object, already when being created. Try this by creating a new Car object in the Python shell as:

```
>>> v2 = Car(model="V90")
Initializing..
>>> v2.getModel()
Model: V90
```

Composition

Composition is simply the act of combing multiple classes into a new class interface. The purpose could for instance be to build functionality in layers that could be tested separately, before combining them into one interface for new purposes. In the following example, Vehicle and Car are combined into a new class called Transport which does little more then to provide access to members of both of the other classes.

```
from vehicle import Vehicle
from car import Car

class Transport():

    def __init__(self):
        self.vehicle = Vehicle()
        self.car = Car(model="V90")

if __name__ == "__main__":
    t = Transport()
    t.vehicle.getInfo()
    t.car.getModel()
```

```
Initializing..
Initializing..
Vehicle
  |-->type: Undefined
  |-->weight: 0
Model: V90
```

Exercise 2: Manipulating geometry

In the examples under the geometry folder the following files may be found.
main.py – Main module skeleton
polygons.txt – Text file containing polygon definitions
polyplotter.py – Module to plot polygons
test.py – Example of polygon plot

Open and examine the test.py file, then run it and observe the result. The task is write a python script to load the contents of the file polygons.txt, parse the values to obtain the points in of the polygons and to plot the polygons in z-order to produce the same result. For this purpose a skeleton module has been provided as main.py. The z-axis is defined to point towards the observer in front of the screen.

Exercise 2: Solution walk-through

Presented code example

Part II – Scientific python

Python vs MATLAB

See appendix: Numpy for MATLAB users

Arrays and plotting

In the following section, we will be using the IPython console. IPython has many useful features for interactive console use. Start Spyder and IPython. Activate the pylab interface to matplot lib by typing ‘pylab’ in the IPyhton prompt.

1. Numpy array creation

- SciLN 3.1.2 – 3.1.4 : Arrays and types

2. Numpy array selections and methods

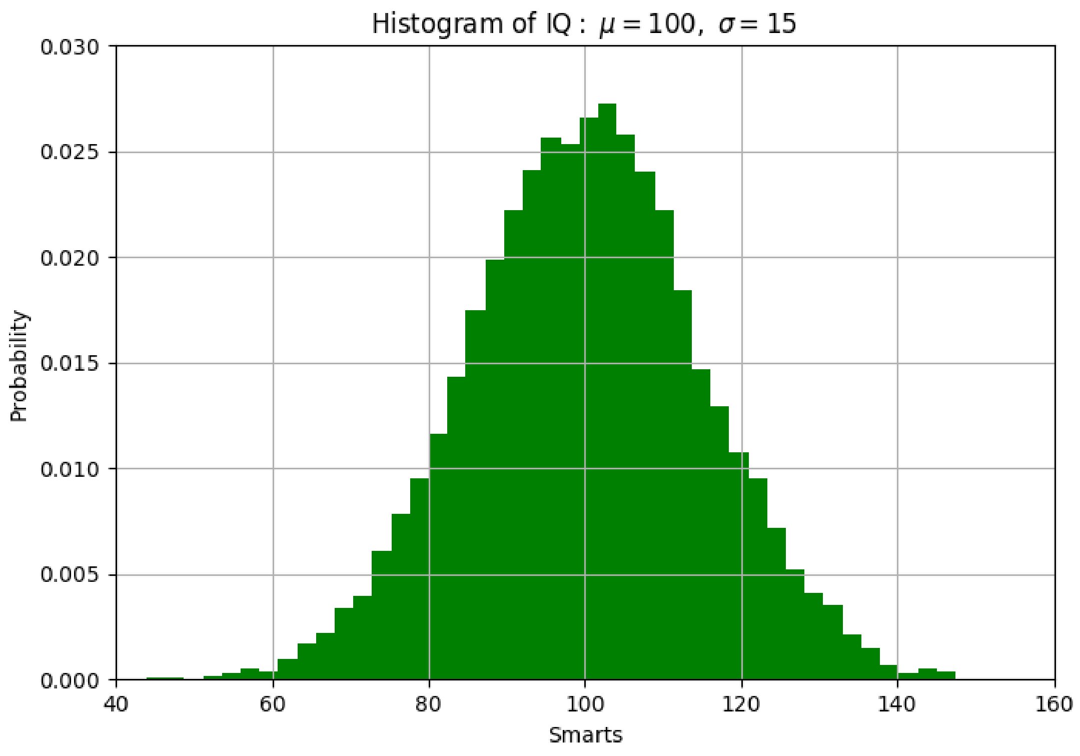
- SciLN 3.1.5 – 3.1.7 : Indexing, slicing, copies and views
- SciLN 3.2.1 : Elementwise operations

3. Plotting arrays

- SciLN 4.2.2 – 4.2.4 : Simple plots
- SciLN 4.6 : Reference - Graph properties

Exercise 3: A histogram from array of samples

Open the example file histogram.py, and run it in Spyder. The code generates a numpy array ‘x’ which is a random variable that consists of IQ test samples from a population of 10 000 individuals. The average IQ in the population is assumed to be 100 and the standard deviation is 15. Given this example array, complete the code to plot a histogram similar to the example image below.



Exercise 3: Solution walk-through

Presented solution example.