How I started Learning Python

From a student. Around 2010 when a student, Joseph Cali, in my Elec6710 Semiconductor Devices class turned in his homework / project written in python.

I always had an interest in computers and programming and experimented with various languages in the past. I was attracted by the beauty of language syntax and started learning it and using it for teaching and research. Joseph introduced me to the basics and gave me a few links to get me started. He also very patiently answered many of my beginner's questions, taught me how to compile packages from source in local folders which I had to do as I do not have super user right in my lab linux machines, introduced me to sphinx for documentation which I use for making the class sites.

Joseph also wrote a program to process data files from Synopsys Sentaurus TCAD, which I rewrote during the winter break of 2011, and added a Graphical user interface using Qt.

I have also written many interactive programs designed to teach digital electronics (Elec 2210) and semiconductor devices (5700, 6700, 6710, 7710).

I was attracted by:

- Easy to read code
- Forced indentation - which many may not like
- It is dynamically and strongly typed
- It has very good memory management so I can focus on writing codes and not worry (too much) about allocating and releasing memory
- Easy object oriented programming
- Strong text processing - often we need to deal with data, but data is in text files in particular format of a given CAD tool
Function is also data - it supports some functional programming
Numerous codes / packages available
It is free, I was doing remote ssh tunneling to access on-campus Matlab license, I am pretty good at automating it, so not a big deal for me personally, but it made it much easier for the students
Much of what I like about Matlab can also be done in better ways
It allowed me to do "advanced" programming with much less effort than what I used to do in C
Graphics (using matplotlib) is beautiful, better than Matlab in some ways

Install Python:

See my earlier notes on installing the EPD python distribution.

This can be done for windows and linux more easily than for Mac.

Mac came with python, however, you may need to install packages.

For this introduction, I assume you are using windows, the EPD distribution, python 2.7.

Also install pyscripter (just google it), a light weight IDE.

Install Wing IED 101, the free version.

Each IDE has its strength, it is good to have an extra IDE in hand.

**import Statement - use existing codes**

There are many "packages" of routines already written. So you do not need to re-invent the wheel. We need to import them so we can use them.
import numpy as np
from myplot import myplot
from matplotlib.pyplot import show

x = np.linspace(0, 10, 100)
y = np.sin(x)

myplot(x=x, y=y, xlabel='x', ylabel='y', color='red')
show()

You see a number of ways to import modules. We have imported numpy, but did not say which functions we want to import.

We imported the "myplot" function from "myplot" module, a module I wrote (you have been using) to do flexible plotting in one line.

We have imported "show" function from "matplotlib.pyplot", matplotlib is a package which contains many modules, pyplot is one of them.

Click the run icon

You should see a graph:
Let us look at them line by line.

```python
x = np.linspace(0, 10, 100)
```

This creates a variable "x", we assign "x" a value, that is returned by the function linspace from the np module (np.linspace), that function has 3 arguments, a start, an end, and the number of points. Pretty much it does the same job as Matlab's linspace function.

```python
y = np.sin(x)
```

This creates a new variable "y", which is assigned a value np.sin(x).

```python
myplot(x=x, y=y, xlabel='x', ylabel = 'y', color='red')
```

Here we plot y as a function of x, with labels for x and y-axes, and color of line all specified in just one line. There a lot more options you can use for myplot, but this should get you started with plotting.
The prefix "np." is necessary as we may have "sin(x)" function in different modules, we need to use the module name to differentiate them. For instance, there is "sin(x)" in the math module as well, but that function does not operate on an array.

**Interactive Python Shell**

You can do quick testing of ideas within the interactive shell. For anything serious, you want to put your codes in a .py file.

The interactive shell can be an excellent way of learning python basics though.

Let us do some simple math.

```python
>>> 1 + 10
11
>>> 1 - 10
-9
>>> 1 * 10
10
>>> 1 / 10
0
```

What is wrong? 1/10 is 0 ! This is the default behavior, integer 1 is divided by integer 10, result is rounded down to give also an integer.

Often we want really 1/10 to be 0.1. This is a common error. To do that, either use float numbers:

```python
>>> 1.0 / 10
0.1
```

Or:
Variable and Assignment

```python
>>> x = 10
>>> x
10
```

This is different from many other languages. "x" is just a variable. "=" means assignment. "x" is assigned the value of "10", an integer, actually, it is just assigned a reference to an integer "10" that is created in memory. Use the id function to find out more:

```python
>>> x
10
>>> id(x)
30259468
```

This is very different from say in Matlab where the variable actually "holds" the value "10" and you talk about an integer variable or a string variable.

You can understand id(x) as the address of the "object", the integer "10" that was just created.

We can then use "x" as in:

```python
>>> x = 10
>>> x
```
10
>>> id(x)
30259468
>>> y = x + 1
>>> y
11

🌟 Variables are not typed! Objects referenced by variables are.

You may now think the variable "x" is an integer variable coming from a statically typed language, such as C.

Wrong.

The object, "10", referenced by "x", is typed, or is integer type.

>>> x
10
>>> id(x)
30259468
>>> x = 'war eagle!
>>> x
'war eagle!
>>> id(x)
105245440

X is assigned an integer "10" first, which has an ID (understand it as memory address for practical purpose) of 30259468.

X is then assigned a string, and that string is created in a different memory space, as shown by the different ID.
Assignment simply creates a reference to an object, or binds the variable "x" to the value (or object), first "10", then "war eagle!". The object has type, but the variable itself has no type.

What happens to the memory space used by "10"?

It becomes garbage, and python handles the garbage collection automatically by tracking the use of objects in your program.

You must assign a value to a variable before you use it.

Python is dynamically typed (the compiler knows which type of object a variable points to), but also a strongly typed language (restrictive about how types can be intermingled).

Python Objects

All objects have:
1. Identity
2. Type
3. Value

It is often very important to check all 3 of them when you stuck debugging a non-working program. How can we find them out?

```python
>>> x = 10
>>> x  # find value
10
>>> id(x)  # find identity
30259468
>>> type(x)  # find type
<type 'int'>
```
The '#' means everything after it is just "comment" and does not get read by python.

I use them as a way to show you the purpose of each "command"

🌟 The id(object) function:
The function used above is extremely useful, but not well known.

id(obj) returns the "identity" of the object "obj".

This identity is an integer which is unique and constant for this object during its lifetime. It is perhaps the closest to the memory address of an object. You can in fact understand it that way without problems.

Two different objects with non-overlapping lifetimes may have the same identity.

Basic Data Types

1. Integer
2. Floats (floating point real number)
3. Strings - just pieces of text
4. Boolean
5. Complex number

Advanced Data Types for Collection of Data Elements

1. List
2. Tuple
3. Dictionary
None - the Null object

NoneType is a special type. It has just one value, None, kind of like the C value of NULL.
Code blocks are identified by indentation instead of symbols like curly braces commonly found in other languages.

For example,

d2 = {'alan': 90, 'andrew': 100, 'jeremy': 110}

for key, value in d2.items():
    print '%s scored %d points' % (key, value)
List:

Creating and assigning list
```python
>>> list_of_students = ['patrick', 'brian', 'alan']
```
```python
>>> list_of_students
['patrick', 'brian', 'alan']
```
```python
>>> type(list_of_students)
<type 'list'>
```
```python
>>> id(list_of_students)
124077880
```
```python
>>> list_alias = list_of_students
>>> id(list_alias)
124077880
```
```python
Note how we create a list with "[ ]".
```
We also created another variable, list_alias, which points to the same list object, as their IDs are the same.
```
To access elements of a list, use indexing and slicing operations.
Access values in list by indexing

>>> x = [1, 2, 3, 4]

>>> x[0]
1

>>> x[1]
2

>>> x[2]
3

>>> x[3]
4

>>> x[4]
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
IndexError: list index out of range

So the index starts from 0. You may need to get used to this depending your background.

An index error occurs when you do x[4].

Negative index is also handy:

To get the last and next to last elements:

>>> x[-1]
4

>>> x[-2]
Slicing:

```python
>>> x[0:2]
[1, 2]
```

The first index is inclusive, while the 2nd index is exclusive.

Add List Element:

```python
>>> x.append(5)
>>> x.append(100)
>>> x
[1, 2, 3, 4, 5, 100]
```

Remove List Elements

```python
>>> del x[-1]
>>> x
[1, 2, 3, 4, 5]
```

To remove a specific element:
>>> x.remove(3)

To reverse a list:

>>> x.reverse()

To sort a list:

>>> x = [1, 3, 4, 10, 9, 7]

Never, ever do this:

>>> y = x.sort()
>>> print y
None

The list sort method - think of it as a function that does not return a list, it returns the special None value.

The list methods change the value of the list.

**Change values of list elements:**

```python
c >>> x = [1, 2, 3]
c >>> x[0] = 1000
c >>> x
[1000, 2, 3]
```

A list can be made of about anything:

```python
c l1 = ['alan', 'andrew', 'jeremy']
```

A logic representation of how this works is drawn below:
Length of a list:

```python
>>> len(list_x)
2
``` Iteration over a list the python way:

```python
>>> list_a = ['elec2210', 'elec6700', 'elec7710', 'elec8710']
>>> for item in list_a:
...     print item
...elec2210
elec6700
elec7710
elec8710
``` Iteration with index using enumerate:

Often we also need the index, this is best done by using the enumerate function:

```python
>>> for idx, item in enumerate(list_a):
...     print 'no. %d in list_a is %s' % (idx, item)
...no. 0 in list_a is elec2210
no. 1 in list_a is elec6700
no. 2 in list_a is elec7710
no. 3 in list_a is elec8710
```
no. 0 in list_a is elec2210
no. 1 in list_a is elec6700
no. 2 in list_a is elec7710
no. 3 in list_a is elec8710

-Starred Iteration over multiple lists at the same time:

```python
>>> list_1 = [1, 2, 3]
>>> list_2 = ['one', 'two', 'three']
>>> for item_1, item_2 in zip(list_1, list_2):
...     print item_1, item_2
...
1 one
2 two
3 three
```

-Starred Iteration over multiple lists at the same time with index

```python
>>> list_1 = [1, 2, 3]
>>> list_2 = ['one', 'two', 'three']
>>> for idx, (item_1, item_2) in enumerate(zip(list_1, list_2)):
...     print 'no. %d element in list_1 and list_2 are: %s %s' % (idx, item_1, item_2)
...
no. 0 element in list_1 and list_2 are: 1 one
```
no. 1 element in list_1 and list_2 are: 2 two
no. 2 element in list_1 and list_2 are: 3 three

>>> Iteration over multiple lists efficiently (real long lists)
>>> from itertools import izip, count
>>> for idx, item_1, item2 in izip(count(), list_1, list_2):
...     print 'no. %d element in list_1 and list_2 are: %s %s' % (idx, item_1, item_2)
...
no. 0 element in list_1 and list_2 are: 1 three
no. 1 element in list_1 and list_2 are: 2 three
no. 2 element in list_1 and list_2 are: 3 three

>>> List assignment and Aliasing
Type these in an interactive shell,

>>> list_x = ['alan', 'andrew', 'jeremy']
>>> list_y = list_x
>>> for list_item in list_x:
...     print list_item
...
alan
andrew
jeremy

>>>
Here "list_y" actually points to or references the same list object in memory, it is just an alias to list_x, as shown below:

If we remove an element from list_x, it is also removed from list_y, because they point to the same physical object:

```python
>>> list_x.remove('alan')
>>> list_x
['andrew', 'jeremy']
>>> list_y
['andrew', 'jeremy']
```
No. 1 in list_x is jeremy

>>> 

The object referencing situation at this point looks like this:

```
$ python
--snip--
>>> list_x = list(["andrew", "jeremy"])
>>> idx = 1
>>> list_item = list_x[idx]
>>> list_y = list_x
>>> list_x is list_y
True
```

What is important here is to understand what is done by python when it sees the statement "list_y = list_x":

- It first **evaluates** "list_x", the value of which is the list object referenced by "list_x"
- The "=" **assignment** causes python to create another reference to that object, yes, it is the same object. The "list_y" variable is then created, and points to the same list object.

**How do I know if I am dealing with the same list object?**

Simple.

>>> list_x is list_y
True
Or

```python
>>> id(list_x)
51370704
>>> id(list_y)
51370704
```  

Clearly the two variables have the same identity, that is, they reference the same object.

Therefore, you must be extra careful when assigning a list to another, as it is just creating an alias. You may also just avoid using it.

⭐ Cloning a list or copying a list:

```python
>>> list_z = list_x[:]
>>> list_z
['andrew', 'jeremy']
>>> list_z is list_x
False
>>> id(list_z)
51194296
>>> id(list_x)
51370704
```  

Now you see the identities are different for list_x and
list_z. We effectively made a copy of the object.

List membership:
Tuples are sequences too, just like lists. The only difference is that tuples are "immutable", meaning they cannot be changed. Tuples do not have methods the way lists do.

Creating and assigning tuple

Separating some values by comma give us tuple:

```python
>>> t1 = 1, 2, 3
>>> t1
(1, 2, 3)
>>> t2 = (1, 2, 3)
>>> t2
(1, 2, 3)
```

They are equivalent, but often you may want to add the parentheses explicitly.

Accessing elements and immutability

```python
>>> t1[0]
1
```
Note we cannot change \texttt{t1[0]} like we could in changing list element value, tuples are said to be immutable.

An important but somewhat peculiar thing to remember is how to create a tuple with only one value,

\begin{verbatim}
>>> t3 = 3
>>> type(t3)
<type 'int'>
>>> t4 = (3,)  # comma is needed here
>>> type(t4)
<type 'tuple'>
>>> t4
(3,)
\end{verbatim}
The comma is what makes it a tuple rather than just a regular integer. This is very important in future learning of the numpy package for numerical computing.

Tuple is very important when we later discuss functions and returns from functions.
Strings:

Basic Usage:

```python
>>> import webbrowser
>>> url = 'http://www.eng.auburn.edu/~niuguof/2210labdev/html'
>>> webbrowser.open(url)
True
```

"url" is a variable, and assigned a string value.

"webbrowser" is a module, which has a open function we can use to open a web site.

```python
>>> import webbrowser
>>> url = 'http://www.eng.auburn.edu/~niuguof/2210labdev/html'
>>> webbrowser.open(url)
True
```

```python
>>> url.find('~')
26
>>> url[0:26]
'http://www.eng.auburn.edu/~'
```

```python
>>> url.find('~')
26
>>> url[0:26]
'http://www.eng.auburn.edu/'
```
We access strings just like accessing lists or tuples.

Strings, however, are immutable:

```python
>>> url.find('.
) 10
>>> url[7:10] = 'com'
Traceback (most recent call last):
  File "<interactive input>", line 1, in
<module>
TypeError: 'str' object does not support item assignment
```}

**String Formatting for Printing:**

In computers, we can only use finite number of digits to represent numbers. So the numbers inside our computers are approximations. To see their values, we need to print them out using the "print" command.

```python
>>> x = 1.0
>>> x
1.0
>>> print x
1.0
```
Now, everything is expected.

Well, not quite, let us use more digits through formatting strings:

```python
>>> x = 3.3
>>> print 'x=%20.18f' % x
x=3.299999999999999822
>>> print 'x=%25.23f' % x
x=3.29999999999999982236432
```

So 3.3 in computer is not really 3.3, computer finds the closest digital representation of that number available, just like in an analog-to-digital converter!

Let me explain the formatting string `x=%20.18f` % x

The "%" sign is a string formatting operator.

To its left, you place a string.

To its right, you place the value you want to format.

For now this is just a single value.

The `%25.23f` parts of the string are conversion specifiers.
It means the total width is 25, and 23 are used after the
decimal point. That is, the number of decimals is 23.
Both the "width" and "number of decimals" are optional.
The "." has to be there if you want to supply only the
precision.

Count the digits you will see exactly what this means.

These conversion specifies are about the same across all
languages.

If you have multiple values, use tuples:

>>> x = 1.0
>>> y = 10.0
>>> z = x + y

>>> print 'x= %f, y= %f, z=%f' % (x, y, z)
x= 1.000000, y= 10.000000, z=11.000000

Here we did not give specification for precision.

List, strings and dictionaries are three most important
data types. You have learned lists and strings. Now let us
look at dictionaries.
Dictionary: beyond indexing

To create an empty dictionary d1:

```python
>>> d1 = {}
>>> d1
{}
>>> type(d1)
<type 'dict'>
>>> 
```

We can then add value to it, say for each student, I need to store their test score:

```python
>>> d1['andrew'] = 100
>>> d1['alan'] = 90
>>> d1['jeremy'] = 110
>>> d1
{'alan': 90, 'andrew': 100, 'jeremy': 110}
>>> 
```

In the printout,
a. the whole dictionary is enclosed in curly braces {}.
b. Each item has a "key" and a "value"
c. Key is separated from value by a colon (:)

We can of course create a dictionary directly with value like this:

```python
>>> d2 = {'alan': 90, 'andrew': 100, 'jeremy': 110}
>>> d2
{'alan': 90, 'andrew': 100, 'jeremy': 110}
```

In memory, it looks like this:

Internally a hash table is created which allows fast lookup. You search something in google and almost instantly get results for what you search. behind the scene, hash technology is used to look up what you want to search. I'll not get into technical details of hash here. The main point is that lookup by keys is very fast, and to first order, the lookup time is the same for any item in the dictionary.
To see Andrew's score:

```python
>>> d2['andrew']
100
```

To iterate over all values:

```python
d2 = {'alan': 90, 'andrew': 100, 'jeremy': 110}

for key, value in d2.items():
    print '%s scored %d points' % (key, value)
```

The output is:

```python
jeremy scored 110 points
andrew scored 100 points
alan scored 90 points
```

Python's efficient key/value hash table structure is called a "dict". The contents of a dict can be written as a series of key:value pairs within braces { }, e.g. `dict = {key1:value1, key2:value2, ... }`. The "empty dict" is just an empty pair of curly braces {}.

Looking up or setting a value in a dict uses square brackets, e.g. `dict['foo']` looks up the value under the key 'foo'. Strings, numbers, and tuples work as keys, and any type can be a value. Other types may or may not work correctly as keys (strings and tuples work cleanly since they are immutable). Looking up a value which is not in the dict throws a KeyError -- use "in" to check if the key is in the dict, or use `dict.get(key)` which returns the value or None if the key is not present (or `get(key, not-found)` allows you to specify what value to return in the not-found case).

Inserted from `<http://code.google.com/edu/languages/google-python-class/dict-files.html>`
When I have 100 students, it is much easier to look up a person's score from a dictionary than from a list.

In the equisemi module, you will find many uses of dictionary. Here is an example, see if you can find all dictionary uses in this function:

```python
def plot_vg_data(nmos=None, vg_data=None, axes=None, color='red '):
    '''
    plot out MOS data as a function of vg on a set of axes. Each Q / C / V is plotted as a function of Vg.
    if axes does not exist, create the axes. Otherwise, use given axes.
    returns the axes
    '''
    if axes == None:
        axes = {}
    xvar = 'vg'
x = vg_data[xvar]
for k, v in vg_data.items():
    yvar = k
    y = vg_data[yvar]
    line, ax, fig = myplot(x=x, y=y, xlabel=xvar, ylabel=yvar, ax=axes.get(yvar, None), label=nmos.label_suffix, color=color)
    axes[yvar] = ax
return axes
```

We will come back to this later.
Functions

Functions in Python can be very hard to understand, and there are plenty of not so accurate descriptions on how the arguments are passed (pass by reference or pass by value in other languages like C). Neither is exactly right.

To get started, do not think too much about passing by reference or value. Start with simple functions.

Function Definition and Calling Function

```python
import math

def diode_current(isat, vf):
    i = isat * (math.exp(vf/0.0258)-1)
    return i

isat = 1e-15
vf = 0.8
idioide = diode_current(isat, vf)

print 'At Vf = %f V, diode I = %g A assuming isat=%g A' % (vf, idioide, isat)
```

1. The "def" defines the function with its parameters within parentheses and its code indented. "isat" and "vf" in the parentheses are parameters to the diode_current function.

2. Variables defined in the function are local to that function, so the "i" in the above function is separate from a "i" variable in another function. It is thus called local variable.
3. All function parameters are also local variables.

4. The return statement can take an argument, in which case that is the value returned to the caller.

5. "diode_current(isat, vf)" is a function call, isat and vf are called arguments to the function.

6. We do not have to name our function arguments the same as how we name our function parameters in function definition, but often this is a good idea, as it shows the intension of each argument clearly. We could also have directly used numbers as arguments, e.g.

```
diode_current(1e-15, 0.5)
```

There is no difference in final result.

7. This is because each argument is first evaluated, and the resulting object is then assigned to the corresponding function parameter.

```
parameter1 = argument1
parameter2 = argument2
```

You can also say the value of argument 1 is assigned to parameter1 if you prefer. Internally, the object represented by argument1 is bound to parameter1.

You might attempt to say that python passes argument to function parameter by reference at this point. This is not
quite right and can lead to a great deal of confusion down the road.

For now, remember the "parameter = argument" is what happens, or python passes argument to function by assignment.

This should not be surprising - Recall that the usage of variable and assignment are different from in most other languages. So the "pass by value" or "pass by reference" in C does not really apply here.

We will come back to this at a later stage after you are more comfortable with writing functions.

At run time, functions must be defined by the execution of a "def" before they are called.

**Exercise:** explain the difference between a function argument and a function parameter

**Default parameter values and keyword arguments:**

It is a good idea to provide default values for function parameters. For the diode function, we could make 1f the default isat value, 0 as default vf.

```python
def diode_current_2(isat=1e-15, vf=0):
    i = isat * (math.exp(vf/0.0258)-1)
```
return i

vf = 0.8
idiode = diode_current_2(vf = vf)

print 'At Vf = %f V, diode I = %g A using default isat' % (vf, idiode)

In this example, we have given default values to isat and vf. So if we do not give isat argument, it takes on the default value. Also note that we are using "vf=vf" to explicitly assign value to function parameter. This is called keyword argument. The "vf" on the left is function parameter, so it has to be called "vf" as that is how it is defined in the function definition. The "vf" on the right is a variable name, referencing to the object "0.8" earlier with the "vf=0.8" statement. Again, we could have used "x", or "y", but I personally find it more meaningful to use the same name. However, it is important to understand "parameter = argument" is what happens at function call.

Output is:

At Vf = 0.800000 V, diode I = 0.0292749 A using default isat

Return value:

A function can only return one value, like in many other languages.

We can always pack many items into a list, a tuple or a dict.

Accessing Variables in Enclosing Scope:
Here is another example I modified from codes in equisemi.py so it runs standalone.

'''
access variables in the enclosing scope
'''

epox = 3.45e-13
nm_to_cm = 1e-7

def cox(tox_nm=None):
    
    # nm_to_cm and epox are in the enclosing scope
    # so they are accessible here inside the function

    tox = tox_nm * nm_to_cm
    return epox / tox

tox_nm_list = [1, 2, 10]
cox_list = [] # create empty list

for tox_nm in tox_nm_list:
    tmp = cox(tox_nm) # use cox function
    cox_list.append(tmp) # append this to
```python
cox_list
print 'cox is %g F/cm^2 for tox = %f nm' % (tmp, tox_nm)
```

Type or paste this piece of code to your python shell of Wing IDE.
The output of the above code is:

cox is 3.45e-06 F/cm^2 for tox = 1.000000 nm
cox is 1.725e-06 F/cm^2 for tox = 2.000000 nm
cox is 3.45e-07 F/cm^2 for tox = 10.000000 nm

This is also rewritten to show you how to use prior knowledge of creating list and adding elements to it, as well as pythonic iteration using for loop. You also see another example of formatting strings.

⭐ Functions are objects too, and can be passed as argument to another function

''functions are objects too''

```python
def f_double(x):
    tmp = x * 2
    return tmp

def vector_func(func, list_x):
    result = []
    for x in list_x:
        result.append(func(x))
    return result
```

```python
def f_double(x):
    tmp = x * 2
    return tmp

def vector_func(func, list_x):
    result = []
    for x in list_x:
        result.append(func(x))
    return result
```
```python
def make_derivative_func(func, dx):
    
    Generate a new function that is the derivative of a given function func.
    dx is the delta for numerical differentiation.
    
    def derivative(x, **argv):
        return (func(x + dx / 2, **argv) - func(x - dx / 2, **argv)) / dx
```

A function can return another function:

'''
we can use function to create new function
'''

```python
fx = func(x)
result.append(fx)
return result
```

```python
vgs = [1, 2, 3]

vgs_double = vector_func(f_double, vgs)

print vgs
print vgs_double

[1, 2, 3]
[2, 4, 6]
```
dx / 2, **argv)) / dx

    return derivative

import math

def f_test(x):
    return math.sin(x)

df = make_derivative_func(f_test, dx=1e-2)

x = 0

print 'x=', x, ' f_test(x)=', f_test(x)

print 'df(x) =', df(x) # using df just like a function

x = 0  f_test(x)= 0.0
df(x) = 0.999995833339

The highlighted part is called a "doc string", which is similar to comment.

🌟 Immutable Function Parameters:

See if you apply prior knowledge to understand everything that is written, designed to understand how variables can be used.
is output, designed to understand how variables work when passed to function as argument.

... demonstration of function argument passing a ...

def ref_test_1(x):
    id_x_func_initial = id(x)
    print 'x inside ref_test =', x, ' id=', id_x_func_initial
    x = 10  # this is a new assignment, so x is bound to a new object
    print 'now x = 10 has been executed'
    id_x_func_final = id(x)
    print 'x=', x, ' id=', id_x_func_final
    print 'id_x_func_initial == id_x_func_final is:
    print id_x_func_initial == id_x_func_final

    return id_x_func_initial, id_x_func_final

x = 1

id_x_caller_initial = id(x)
print 'x in caller before func call =', x, ' id=', id_x_caller_initial

id_x_func_initial, id_x_func_final = ref_test_1(x)

id_x_caller_final = id(x)

print 'x in caller after func call =', x, ' id=', id_x_caller_final
print '-' * 40

print 'in caller, before func call, %d ' % id_x_caller_initial
print 'now inside func, initial value, %d' % id_x_func_initial
print 'still inside func, final value, %d ' % id_x_func_final
print 'returned to caller, after func call, %d' % id_x_caller_final

Without running the program, see if you can work out the output of the above program yourself on a piece of paper.
Then compare with the output:

Evaluating 32 lines of code...
x in caller before func call = 1  id= 33900248
x inside ref_test = 1  id= 33900248
now x = 10 has been executed
x= 10  id= 33900140
id_x_func_initial == id_x_func_final is: False
x in caller after func call = 1  id= 33900248

in caller, before func call, 33900248
now inside func, initial value, 33900248
still inside func, final value, 33900140
returned to caller, after func call, 33900248

So initially, the "x" inside the function references the same object as the "x" in the caller.

But we have another "assignment" inside the function, which binds the "x" inside the function to another object, as confirmed by the identity change.

However, once we return to the caller, the "x" in the caller still references the same object it referenced before the function call.
Mutable function parameter (e.g. a list):

```python
def func_with_list_parameter(list_a):
    print(list_a)
    print("id(list_a) inside function = %d" % id(list_a))
    list_a = [1, 2, 3]
    print("id(list_a) inside function = %d" % id(list_a))

list_main = ['a', 'b', 'c']

print("id(list_main) = %d" % id(list_main))
print("now calling func_with_list_parameter(list_main)"
func_with_list_parameter(list_main)
print("now I have come out of function")
print("id(list_main) = %d" % id(list_main))
print(list_main)
```

Pasting the above code into your python shell:

```
id(list_main) = 40489168
now calling func_with_list_parameter(list_main)
['a', 'b', 'c']
id(list_a) inside function = 40489168
id(list_a) inside function = 40488192
now I have come out of function
id(list_main) = 40489168
['a', 'b', 'c']
```

The value of list_main has not changed in this case, even though a list is mutable.
```python
def func_modify_list(list_para):
    print(id(list_para))
    list_para += [1, 2, 3]
    print(id(list_para))

list_main = [1, 2, 3]

print(list_main)
print(id(list_main))

func_modify_list(list_main)

print(list_main)
print(id(list_main))
```

Pasting the above code into your python shell:

```
[1, 2, 3]
40488192
40488192
40488192
[1, 2, 3, 1, 2, 3]
40488192
```

This time, the identity remains the same outside and inside the function. However, the list was extended with 3 new elements! In other words, the value of the object referenced by the function list parameter has changed. That change remains effective after the function call.

A bit comment on the "list_para += [1, 2, 3]". The "+=" operator...
A bit comment on the `list_para += [1, 2, 3]`. The `+=` operator is sometimes called "augmented assignment", which really is not true for list. Some often say "x += y" means "x = x + y", which, however, is completely incorrect for list!

For list, there is no "new assignment", which means "creating a new object and then binding the variable to it".

What happens to list is that "+=" simply extended (caution, not append!) the original list's content by another list, as you have seen from the print result.

If this does not convince you, try the following:

```python
def func_is_list_modified_or_not(list_para):
    print id(list_para)
    list_para = list_para + [1, 2, 3]
    print 'inside func', list_para
    print id(list_para)

list_main = [1, 2, 3]
print list_main
print id(list_main)
func_is_list_modified_or_not(list_main)
print list_main
print id(list_main)
```

So what do you think? Is list_main changed or not by the function call given that we have "list_para = list_para + [1, 2, 3]". In fact, that is the only line of code I have changed.
The result clearly shows a change in identity after

```python
list_para = list_para + [1, 2, 3]
```

is executed.

This is easy to understand if you recall what "assignment", that is, the "=" sign does!

The right hand side is evaluated, that is, a new list object is created, and bound to the local variation list_para inside the function.

The original object that was referenced by the previous list_para variable before this assignment, however, is not affected by this.

Consequently, list_main still has the same value after the function call.

Confused slightly? May be? All of this can be understood by inspecting the identity and value of the objects as well.
inspecting the identity and value of the objects, as well as remembering what "assignment" does, and what variable is in python.

If this is your first language, do not worry about the "pass by value" or "pass by argument" - rather, just remember that:

`parameter = argument`

is what happens with a function call!

In this sense, you may say "pass by assignment". Still you need to first correctly understand variable, object, and assignment.
Let us continue with our oxide example with an objected oriented programming (OOP).

We will continue to use the existing cox(tox_nm) function we created earlier.

The basic idea is to pack methods, e.g. cox(tox_nm) and data the methods operate on, e.g. tox_nm, together. There are many advantages to this I will not get into for now. Rather I want to show you an example taken out (and modified for illustration purpose) of the equisemi module I wrote.

⭐ Class and object by example

epox = 3.45e-13
nm_to_cm = 1e-7

class Oxide(object):
    def __init__(self, tox_nm=1):
        self.tox_nm = tox_nm
        self.analyze()

    def analyze(self):
self.cox = cox(self.tox_nm) # defined in enclosing scope

def __str__(self):
    tox_nm = self.tox_nm
cox = self.cox
    s = 'tox = %f nm, cox = %g F/cm^2' % (tox_nm, cox)
    return s

def cox(tox_nm=1):
    '''
    cox for given tox_nm
    '''

    # nm_to_cm and epox are in the enclosing scope
    # so they are accessible here inside the function

    tox = tox_nm * nm_to_cm
    return epox / tox

tox_nm = 1 #nm

oxide = Oxide(tox_nm)

print oxide.tox_nm
print oxide.cox

# better yet

print oxide

Result of running is:

1
3.45e-06
tox = 1.000000 nm, cox = 3.45e-06 F/cm^2

After running this, type in the interactive shell:

```python
>>> tox_nm
1
```

```python
>>> oxide.tox_nm
1
```

```python
>>> oxide.cox
3.45e-06
```

```python
>>> print cox
<function cox at 0x029B6170>
```
Note that python is case sensitive. It is custom to capitalize the first letter of the class name.

Does everything make sense to you? Check out the value, id and type of various variables, including function, to get a good understanding.

Class definition, creation and initialization, the "self" business

"class" defines a class, Oxide, which inherits from a python object.

The "__init__" function is an initialization function that is called when an instance of Oxide is created.
It takes two arguments, self, and tox_nm. "self.tox_nm" is first created, self actually means the instance object itself.

The "def" of "__init__" function is:
__init__(self, tox_nm=1)

What actually happens during instantiation is that python calls the "__init__" function of Class Oxide, the parameter "self" is given the value of the object being initiated.

To create an Oxide object,
oxide = Oxide(tox_nm)

The variable "oxide" is created, and bound to an Oxide object. We call this an instance of the class Oxide. The process is called class instantiation. The syntax is the same as calling a function.

The "__init__" function is called during instantiation.

When oxide = Oxide(tox_nm) is run, it will call the "__init__" function:
Oxide.__init__(self = oxide, tox_nm = tox_nm)

This is why the first argument to a class method (a
function inside a class) is called commonly as "self", as the name implies the meaning of this function argument.

It can be called anything else, but this is not recommended.

The "__init__" function then calls "self.analyze()", which is defined next.

In the function analyze(), a new variable "self.cox" is created, and calculated using the cox function we defined before.

We could move it inside the class if we want. We choose not to move it inside so the original codes using it do not need to be changed.

We often create many different Oxide objects, or instances. The self parameter differentiates them.

Next, let us create two Oxide instances:

oxide_1 = Oxide(1)
oxide_2 = Oxide(2)

We then call the Oxide.__str__(self) function on both instances, and print them out:

str1 = Oxide.__str__(oxide_1)
print str1

gstr2 = Oxide.__str__(oxide_2)
print str2

Output is:

tox = 1.000000 nm, cox = 3.45e-06 F/cm^2
tox = 2.000000 nm, cox = 1.725e-06 F/cm^2

This should explain why you see "self" as the first argument to your functions in the class - also called class methods.

⭐ Printing your own type object like you print a built-in type

Recall that the Oxide.__str__(self) function returns a string that has information, which we may call "values" of the oxide object in question, self.

"__init__" and "__str__", as you may have guessed, are special methods python recognizes, we have basically overloaded the default object methods with the same name.

Well, I have used these to explain "self" mainly. Not just that. This also underlies why we can print an oxide object just like it is a built-in integer or string object. Internally, it
calls the "__str__" function of the Class in question.

```python
>>> print oxide_1

tox = 1.000000 nm, cox = 3.45e-06 F/cm^2
```

Very nice, right? You can now create your own type of object and print it like a built-in type object.

**What to do now:**

Now you should be in a position to understand the Semi, Nsemi, Psemi, Nmos classes in my equisemi module. They use more advanced features which you can get by without using.

We could also change the "tox_nm" value of oxide_1, and then call the "analyze()" method to update "cox" etc.

```python
>>> oxide_1.tox_nm = 20

>>> oxide_1.analyze()

>>> print oxide_1

tox = 20.000000 nm, cox = 1.725e-07 F/cm^2
```
Updating class member attributes in a better way

We can even write a class method to do this automatically. Let us rewrite our class as follows (I'll not explain everything about it at this stage, but you can use it as a template for your own codes):

class Oixde(object):
    def __init__(self, tox_nim=1):
        self.tox_nim = tox_nim
        self.analyze()

    def analyze(self):
        self.cox = cox(self.tox_nim) # defined in enclosing scope

    def __str__(self):
        tox_nim = self.tox_nim
        cox = self.cox
        s = 'tox = %f nm, cox = %g F/cm^2'
        return s % (tox_nim, cox)

    def update_value(self, **kwargs):
        unknown_keys = []
        for key in kwargs.keys():
            if self.__dict__.has_key(key):
                self.__dict__[key] = kwargs[key]
else:

unknown_keys.append(key)

    if len(unknown_keys) > 0:
        msg = 'Invalid class member(s): %s' % unknown_keys
        raise ValueError(msg)
    self.analyze()

def cox(tox_nm=1):
    '''
    cox for given tox_nm
    '''

    # nm_to_cm and epox are in the enclosing scope
    # so they are accessible here inside the function

    tox = tox_nm * nm_to_cm
    return epox / tox

Now, try:

>>> oxide = Oxide(1)

>>> id(oxide)
>>> oxide.update_value(tox_nm = 5)

>>> print oxide

Tox = 5.000000 nm, cox = 6.9e-07 F/cm^2

>>> oxide.update_value(tox_nm = 50)

>>> print oxide

Tox = 50.000000 nm, cox = 6.9e-08 F/cm^2

You should see from "print" results that with the "update_value" method, we are simply updating the same object.

This can have advantages compared to creating new objects, in terms of both speed and memory usage. Say creating 100,000 objects is more expensive than creating one object and updating its value.

You may now understand why I did not put all the codes in "analyze(self)" in the "__init__" method. I can now use these codes in the "update_value" method. One can in principle to use the "__init__" method in update_value, I prefer to separate them and keep the initialization method simple.

⭐️ Official terminology
The "tox_nm" and "cox" in class Oxe are **data attributes**.

The "__init__", "analyze", and "__str__" functions defined as part of the class Oxide definition are class **methods**.

⭐ **Find out attributes of a class**

In interactive shell, use the `dir(object)` function:

```python
>>> dir(oxide)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'analyze', 'cox', 'tox_nm', 'update_value']
```

I have highlighted the methods (or **method attributes**) we defined explicitly in red, and the **data attributes** in green.

⭐ The built-in "__dict__" member is what was used in the `update_value` method.
Refer to the list page for now, that is enough to do all you need to do.
Python's efficient key/value hash table structure is called a "dict". The contents of a dict can be written as a series of key:value pairs within braces { }, e.g. dict = {key1:value1, key2:value2, ... }. The "empty dict" is just an empty pair of curly braces {}.

Looking up or setting a value in a dict uses square brackets, e.g. dict['foo'] looks up the value under the key 'foo'. Strings, numbers, and tuples work as keys, and any type can be a value. Other types may or may not work correctly as keys (strings and tuples work cleanly since they are immutable). Looking up a value which is not in the dict throws a KeyError -- use "in" to check if the key is in the dict, or use dict.get(key) which returns the value or None if the key is not present (or get(key, not-found) allows you to specify what value to return in the not-found case).

Inserted from <http://code.google.com/edu/languages/google-python-class/dict-files.html>
See the myplot part in the first page.

See the many examples in equisemi.py (all the functions with demo in it)
For now you do not need to write data to files or read data from files.

I will teach you how to do this as we go along.
Id-Vg

Thursday, March 22, 2012
12:17 PM

Id-Vg log scale

Note the low Vds and high Vds difference

In subthreshold, no difference so long as VDS > 3 phi_t
Linear Id - Vgs

Low Vds, Ids - Vgs is linear (we are assuming constant mobility - which is not the case once we consider degradation of mobility with increasing Vgs)
Use more Vds values:
Nsub = 5e17

def id_vg_tox_demo():
    nsub = 5e17
    tox_nm_s = [1, 5, 10]
    vgss = np.linspace(-0.5, 2, 50)
    vsbs = [0]
    vdss = [2]
    ylog = True
    IdVgTox(nsub=nsub,
            tox_nm_s=tox_nm_s,
            vgss=vgss,
            vdss=vdss,
            vsbs=vsbs,
            ylog=ylog,
    )
def id_vg_tox_demo():
    nsub = 1e17
    tox_nm_s = [1, 5, 10]
    vgss = np.linspace(-0.5, 2, 50)
    vsbs = [0]
    vdss = [2]
    ylog = True
    IdVgTox(nsub=nsub,
            tox_nm_s=tox_nm_s,
            vgss=vgss,
            vdss=vdss,
            vsbs=vsbs,
            ylog=ylog,
    )
Your ps0 and psl difference equals VCB only in strong inversion and linear operation region.
def id_vg_parametric_demo():

    # create a ids-vgs plotter object
    ivp = IdVgParametric()

    nsubs = [1e16, 1e17, 5e17, 1e18]

    for nsub in nsubs:
        ivp.nsub = nsub
        ivp.para_sweep(var = 'tox_nm', paras=[1, 2, 10], vdss = [2.0])
def id_vg_nested_sweep_demo():
    # create a ids-vgs plotter object
    ivp = IdVgParametric()

    nsubs = [1e16, 5e17]
    ivp.vds = 2.0

    for nsub in nsubs:
        ivp.nsub = nsub
        ivp.nested_sweep(sweep_parameter = 'tox_nm',
                         sweep_parameter_values = [1, 5],
                         bdbias_name = 'vsb',
                         biases = [0, 2.0])

Source file is in idvg-demo.py

This is nsub=1e16, 5nm shows more Vsb dependence, not surprisingly

Also note that on a log scale, Vsb dependence is much more visible in subthreshold region than in strong inversion region!
Recall how the subthreshold current and strong inversion current relate to threshold voltage? This is one of the main reasons.
NEXT we increase $n_{\text{sub}}$ to $5 \times 10^{17}$, the $V_{\text{sb}}$ dependence (color comparison) is a lot stronger!

Of course, the dependence is stronger for thicker $t_{\text{ox}}$. 
We can also make the comparison another way:

def id_vg_nested_sweep_demo_2():
    # create a ids-vgs plotter object
    ivp = IdVgParametric()
    tox_nm_s = [1, 5]
    ivp.vds = 2.0
    for tox_nm in tox_nm_s:
        ivp.tox_nm = tox_nm
        ivp.nested_sweep(sweep_parameter = 'nsub',
                         sweep_parameter_values = [1e16, 5e17],
                         bdbias_name = 'vsb',
                         biases = [0, 2.0])

def main():
    # id_vg_demo()
    # interactive_id_vg_demo()
    # id_vg_tox_demo()
    # id_vg_parametric_demo()
    # id_vg_nested_sweep_demo()
    id_vg_nested_sweep_demo_2()
The figure above shows 1nm Vsb dependence for two substrate doping levels.

Again, it is clear that Vsb dependence is stronger at higher doping, but overall it is small because of thin oxide.

This is for 5nm oxide, Vsb dependence is weak at 1e16.
doping, but a lot stronger at $5 \times 10^{17}$ doping.
def id_vg_demo():
    nsub = 1e16
    tox_nm = 5
    vgss = np.linspace(-0.5, 2, 50)
    # vdss = [2]
    vsbs = [0]
    ylog = False
    IdVg(nsub=nsub,
         tox_nm=tox_nm,
         vgss=vgss,
         vsbs=vsbs,
#         vdss=vdss,
         ylog=ylog,
         )

def id_vg_tox_demo():
    nsub = 5e17
    tox_nm_s = [1, 5, 10]
    vgss = np.linspace(-0.5, 2, 50)
    vsbs = [0]
    vdss = [2]
    ylog = True
    IdVgTox(nsub=nsub,
             tox_nm_s=tox_nm_s,
             vgss=vgss,
             vsbs=vsbs,
             vdss=vdss,
             ylog=ylog,
             )

def id_vg_parametric_demo():
    # create a ids-vgs plotter object
    ivp = IdVgParametric()

    nsubs = [1e16, 1e17, 5e17, 1e18]

    for nsub in nsubs:
        ivp.nsub = nsub
        ivp.para_sweep(var = 'tox_nm', paras=[1, 2, 10], vdss = [2.0])
...

    # using default Vds values for id-vgs sweep
### Nested IdVg parametric sweep.

Sweep Paras: sweep tox_nm and nsub.

Either can be outer-para sweep or inner-para sweep, then select either varying Vds or Vsb for Id-Vg curve.

Sample codes for a new figure for each nsub (outer_para), and in each plot, two inner_para are used (tox_nm), for each inner_para (tox_nm), two vsbs are plotted (0, 2.0), vds is set to 2.0V initially:

```python
class IdVgNestedParametric(object):
    
    Nested IdVg parametric sweep.

    Sweep Paras: sweep tox_nm and nsub.

    Either can be outer-para sweep or inner-para sweep, then select either varying Vds or Vsb for Id-Vg curve.

    Sample codes for a new figure for each nsub (outer_para), and in each plot, two inner_para are used (tox_nm), for each inner_para (tox_nm), two vsbs are plotted (0, 2.0), vds is set to 2.0V initially:

    ivp = IdVgNestedParametric()
    ivp.vds = 2.0
    ivp.set_outer_bias('vsb', [0, 2.0])
```
ivp.set_outer_para('nsub', [1e16, 5e17])
ivp.set_inner_para('tox_nM', [1, 5])
ivp.sweep()

```python
def __init__(self, tox_nM = 2,
             nsub = 1e16,
             vgss = None,
             vds = 0.1,
             vsb = 0):
    self.tox_nM = tox_nM
    self.nsub = nsub
    self.vgss = vgss
    self.vds = vds
    self.vsb = vsb
    self.ivp = IdVgParametric(tox_nM = tox_nM,
                               nsub = nsub,
                               vds = vds,
                               vsb = vsb)
    self.outer_para = {}
    self.inner_para = {}  

def set_outer_para(self, name, values):
    self.outer_para[name] = name
    self.outer_para['values'] = values

def set_inner_para(self, name, values):
    self.inner_para[name] = name
    self.inner_para['values'] = values

def set_aux_bias(self, name, values):
    self.ivp.aux_bias[name] = name
    self.ivp.aux_bias['values'] = values

def sweep(self):
    outer_var = self.outer_para[name]
    outer_var_values = self.outer_para['values']
    inner_var = self.inner_para[name]
    inner_var_values = self.inner_para['values']
    ivp = self.ivp

    for outer_var_value in outer_var_values:
        ivp.para[outer_var] = outer_var_value
        ivp.sweep_param[name] = inner_var
        ivp.sweep_param['values'] = inner_var_values
        ivp.sweep()

def id_vg_nested_parametric_demo():
    ivnp = IdVgNestedParametric()
    ivnp.vds = 2.0
    ivnp.set_aux_bias('vsb', [0, 2.0])
```
ivnp.set_outer_para('nsub', [1e16, 5e17])
ivnp.set_inner_para('tox_nm', [1, 5])
ivnp.sweep()

def interactive_id_vg_demo():
    vsbs = [0, 1, 2]
    vgss = np.linspace(-0.5, 1.5, 50)
    InteractiveIdVg(vgss=vgss, vsbs=vsbs, ylog=True)

def main():
    # id_vg_demo()
    # interactive_id_vg_demo()
    # id_vg_tox_demo()
    # id_vg_parametric_demo()
    # id_vg_nested_sweep_demo_2()
    id_vg_nested_parametric_demo()

if __name__ == '__main__':
    main()
    show()