Material 1

Material relevant for the numbered Competence and Assignment.

imports

```
import numpy # basic format; everything in numpy library now available
import numpy as np # typical use.  same as above, but now called np
from numpy import cos, sin, tan, pi # import specific parts of numpy
```

Type `np.[TAB]` to see what else `numpy` can do, and e.g. `np.random.[TAB]` to explore this part of `numpy`.

complex numbers

```
z = 3 + 4j # Engineers like j for the complex unit
```

comparison & assignment

Written mathematics can be sloppy. What does $x = 2$ mean? Does it mean *assign the value 2 to* x? or does is ask a question: *does* x *equal 2?* Programming must be more precise. Python uses = for the former and == for the latter.

```
thing = 2 # assign the value of 2 to a variable called "thing"
thing == 2 # compare the value of "thing" to the number 2
thing > 2, thing >= 2, thing < 2, thing <= 2 # various comparisons
thing != 2 # this means "not equals to"
```

divison

We've met $+$, $-$, $\times$, and $/$. We also have *integer division* `//` and *modulo division* `%`:

```
13//3, 13%3 # 13/3 = 4 times 3 remainder 1
```

if statements

Make *decisions* based on comparisons:

```
if thing > 2:
    print("thing is larger than 2") # print displays to the user
else:
    print("thing is not larger than 2") # do this if the condition above is not true
```

Combine conditions with `and`, `or`, `not` e.g. `((a < 2) and (b < 2)) or (a*b < 4)`

loops

Do something multiple times for different values

```
for n in [1,2,3]:
    print(n)

for n in range(10): # range makes values when needed
    print(n)
```

Also, loops based on a comparison:

```
n = 0
while n < 10:
    n += 1 # shorthand for n = n + 1, i.e assign the value of (n+1) to the variable called n
```

or an infinite loop:

```
while True:
    # do something
```

indentation

Python uses *indentation* (i.e. tabs) to control the flow of code.
Indenting lines makes them "belong" to whatever started the block, e.g. `for`, `def`, `if`, etc.
This can be *nested* like so:

```
for n in range(10):
    if n > 5:
        print(n, " is bigger than 5")
```

functions

```
def func_name(var1,var2,var3=3): # var3 has a a default value and is therefore optional
    """Human-readable description of what the function does.
    See PEP-257"""
    # arbitrary code to process variables var1, var2, var3
    output = var1 + var2 + var3
    return output # ends with a return statement

thing = func_name(1,2)
```

Functions are *first class* in Python, which means you can assign them as you would an integer or any other type. e.g.

```
def f(x):
    return x**2

g = f
g(3)
```

```
9
```

functions and returning multiple things

Functions can have multiple inputs and multiple outputs.

```
def func_name(var1,var2,var3):
    return var1+var2+var3, var1*var2*var3, var1**2+var2**2+var3**2

a,b,c = func_name(3,6,9)
```

flow control extras

e.g. display every odd integer with square less than 400

```
for n in range(100):
    if n % 2 == 0:
        continue # skip the following and start the next iteration
    if n**2 > 400:
        break # exit from the for loop
    print(n)
```

building lists

Build a list of the integers from zero to nine.

```
stuff = list() # make a new, empty list
stuff = [] # another way to make a list
for n in range(10):
    stuff.append(n) # append n to the end of the list

stuff = [n for n in range(10)] # list comprehension example
```

arrays vs lists

A Python list can contain anything (even itself!). (Make a list and use the append method to make this happen.)

A `numpy.array` typically contains lots of the same thing, e.g. lots of integers or lots of real numbers.

Opertions on an array are very very fast. Operations of lists are typically rather slow. Operations such as `x**2` work if `x` is an array, but not if it's a list.

Newton's method case study

See Newton's method.

```
def newton(f, x0, fprime=None, tol=1e-6, maxiter=50):
    if fprime is None:
        def fprime(x):
            h = 1e-6
            return (f(x+h)-f(x))/h
    x = x0
    for n in range(maxiter):
        x -= f(x)/fprime(x)
        if abs(f(x)) < tol:
            return x
    return x
```

`f` is a Python function which takes one argument `x0` is the initial guess at the solution. `fprime` is a function which gives the differential; if this is not given then it will be approximated.

```
def func1(x):
    from numpy import tan
    return tan(x)-1

sol = newton(func1, 0.5)
sol
```

```
0.785398163749
```

Newton's method from `scipy`

If you want to solve a real problem, then you probably want to use a library rather than writing your own implementation. e.g. `scipy` offers a `newton` method:

```
from scipy.optimize import newton
def func(x):
    return x**2 - 2

answer = newton(func, 1)
```

Plotting

```
import matplotlib.pyplot as plt # import a specific part of the library and call it plt

import numpy as np
x = np.linspace(-1,1,101)
y = x**3 - 2*x**2

plt.plot(x,y)
plt.title("This is the title")
plt.xlabel("Give x-axis a label")
plt.ylim([-1,1])
plt.savefig("plot.pdf")
plt.close('all')
```