6 Python

In this course we will be using python. Python is a programming language. It's freely available and very widely used, especially in the scientific community. There are many other programming languages that can do everything that python can do, sometimes better, sometimes worse. Which programming language to choose depends on the problem that you are trying to solve. It also depends if you already have parts of the program, for example given to you by your supervisor.

The basic concepts are pretty much the same in every programming language. You only need to learn the new *syntax* when you switch to a different language. Once you know how to program, learning a new language is only a matter of hours or days.

For example, here are two programs in C and python which output the sum of $a + b$. They do the same thing, only the syntax is different.

```
a = 1
b = 2
print a+b
```

Code 31: Python program printing the sum of two numbers.

```
int main (){
        int a = 1;
        int b = 2;
        printf("%d",a+b);
        return 0;
}
```

Code 32: C program printing the sum of two numbers.

As you can see by just looking at the above examples, python is pretty easy to understand and learn.

There are some common construct in programming languages that will appear over and over again. It's important that you understand them. We'll go over them in the remainder of this section.

6.1 Variables

Variables can store information in the same way as the registers in our Assembler language. However, variables have names and are thus much easier to work with. One variable can either store a single number, a string, a memory address, a picture or anything else. What you do with a variable depends on the context. In python, you do not have to explicitly declare a type of a variable (i.e. a picture vs a number). You need to know what kind of information the variable hold. For example, it makes sense to add 2 to a number, but it does not make sense to add 2 to a picture. In the above example, you can see that in C we need to define the type of a variable explicitly.

In python you assign a value to a variable with the equal sign (see examples above).

6.2 Conditional statement

One of the most common constructs is a conditional statements. The computer executes a command only if a certain condition is met.

```
if a==1:
        print "a is one"
else:
        print "a is not one"
```

Code 33: Conditional statement in python.

In this example, the condition is "is a equal to 1". Note that the *is equal to* part is written using two equal signs in python. This is an important thing to keep in mind. If you use one equal sign, you assign the value on the right to the variable on the left. With two equal signs, the value on the left is compared to the value on the right. These are fundamentally different operators and you need to make sure you use the correct one each time.

If the condition "the variable a is equal to 1" is met, the computer prints a message "a is one" onto the screen. If the condition is not met, the computer executes only the commands after the `else` keyword. In this case, the computer prints "a is not one" on the screen.

In python you group the commands after the `if` and `else` statements by using spaces or tabs. Thus, we could easily output multiple lines (the second line is the German translation of the first line) like this:

```
if a==1:
        print "a is one"
        print "a ist eins"
else:
        print "a is not one"
        print "a ist nicht eins"
print "this line is printed in either case"
```

Code 34: Conditional statement in python.

6.3 Loops

Loops are constructs which allow you to perform the same command multiple times (i.e. in a loop). This is a vital construct in any programming language and incredibly helpful as you do not have to write the same code multiple times. For example, the following code snippet will output the numbers from 0 to 9, but you only need to write one print statement

```
a = 0
while a<10:
        print a
        a = a + 1
```

Code 35: Loop statement in python.

The last line increments the variable a. You need to read the right side of the equal sign first. For example, if the variable a currently holds the value 1, then the computer takes this value and adds 1 to it. The results is 2. Only then it stores this value back into the variable a. Compare this to the Assemnbler statement `ADD`, where the register the value is stored to comes last.

Another way to get the same result is a *for loop*:

```
for i in range(10):
        print i
```

Code 36: For loop in python.

The `range` statement creates a list of number to iterate over. Instead of using `range` one can use any list or array (see next section).

6.4 Arrays

We will often come across arrays, a list of number. Arrays are especially important when we work with data. In python arrays make use of square brackets `[ ]`. The following code shows you a few example how to define, initialize, manipulate and loop over arrays. Note that the first element in any array has the index 0.

```
a = []              # Define an empty array
a.append(1.)        # Add the number 1 to the array
b = [1e-16,1.0]     # Initialize array with a list of numbers
c = [1.] *10        # Initialize each element in an
                    # array of length 10 with 1
c[0] = 2.           # Set the first element to 2
c[0] = a[0]         # Set the first element of c
                    # to the value of a[0]
for e in c:         # Loop over all element of c
        print e     # print the element
print len(c)        # print the length of the array
```

Code 37: Arrays in python.

6.5 Reading files

A lot of the time, we will be reading data from a file and then manipulate it in python. Reading a text file with numbers stored in it is very easy in python. Usually we'll be dealing with floating point numbers in a file, which we read into an array. Suppose we have a datafile with two columns separated by a space. The following code reads this file into two arrays, x and y.

```
x = []                              # initialize arrays
y = []

for line in open("file.txt"):       # loop over lines
    data = line.split(" ")          # split into columns
    x.append(float(data[0]))        # convert to float
    y.append(float(data[1]))        #   and append
```

Code 38: Reading a file in python.

6.6 Writing files

Writing a file works in a similar way. The following code snippet will write the arrays x and y to file (we assume they have the same length).

```
file = open("file.txt", "w")     # Open a writable file
for i in range(len(x)):          # Loop over array
    file.write("%e %e\n" % (x[i],y[i]))
file.close()                     # Close file
```

Code 39: Writing a file in python.

6.7 Functions

Functions allow you to reuse and organize code. Functions have a name can take (multiple) arguments. They can also have a return value. In python functions are declared with the `def` statement. Below is a simple example:

```
def add(a,b):
    return a + b
print add(1,2)    # prints 3
```

Code 40: Functions in python.

7 Matrix Decomposition

7.1 Fibonacci Numbers Take 2

The next big things we'll look at are matrix decomposition, eigenvalues and eigenvectors.

But before we go into the meaty stuff, let's look at the Fibonacci numbers again. We wrote the recursion sequence before as $F_n = F_{n-2} + F_{n-1}$ with $F_0 = 0$ and $F_1 = 1$. Let's write this as a matrix equation! You can easily convince yourself that the following equation will also produce the Fibonacci numbers:

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix}.$$

We can furthermore introduce a vector and write it as

$$\vec{F}_n = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \vec{F}_{n-1}.$$

The starting value is

$$\vec{F}_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

Now that we have a matrix equation we can look at things like eigenvectors and eigenvalues. It'll become clear why we want to do that in a minute.

I'll give you 5 minutes to calculate the eigenvalues and eigenvectors of

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

The answers are:

$$\lambda_1 = \frac{1+\sqrt{5}}{2} \qquad \text{with} \qquad \vec{x}_1 = \begin{pmatrix} \frac{1}{2}(1+\sqrt{5}) \\ 1 \end{pmatrix}$$
$$\lambda_2 = \frac{1-\sqrt{5}}{2} \qquad \text{with} \qquad \vec{x}_2 = \begin{pmatrix} \frac{1}{2}(1-\sqrt{5}) \\ 1 \end{pmatrix}.$$

Let's express the initial value as a combination of eigenvectors, which is simply

$$\vec{F}_0 = \frac{1}{\sqrt{5}}\left(\vec{x}_1 - \vec{x}_2\right).$$

To get the n-th Fibonacci number, we have to apply the matrix A n times to $\vec{F}_0$. Now that we have decomposed F_0 into eigenvectors, the result is very easy to write down as each eigenvector just gets multiplied n times with its eigenvalue.

$$\vec{F}_n = \frac{1}{\sqrt{5}}\left(\lambda_1^n \vec{x}_1 - \lambda_2^n \vec{x}_2\right).$$

or if we only look at the top component of the $\vec{F}$ vector,

$$\begin{aligned} F_n &= \frac{1}{\sqrt{5}}\left(\lambda_1^n \frac{1}{2}(1+\sqrt{5}) - \lambda_2^n \frac{1}{2}(1-\sqrt{5})\right). \\ &= \frac{1}{\sqrt{5}}\left((\frac{1}{2}(1+\sqrt{5}))^{n+1} - (\frac{1}{2}(1-\sqrt{5}))^{n+1}\right). \end{aligned}$$

We now have an anlytic expression for the Fibonacci numbers. However, we need to use complicated functions like the n-th power and the square root to get the Fibonacci numbers with the above algorithm. Especially in floating point precission, this can be inaccurate.

But one advantage of writing the Fibonacci recursion relation in this form is that we can come up with yet another algorithm in $O(\log n)$ instead of $O(n)$. Recall that for matricies

$$A^{2n} = A^n A^n$$

using this together with the result from above, i.e.

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

gives the relations

$$\begin{aligned} F_{2n-1} &= F_n^2 + F_{n-1}^2 \\ F_{2n} &= (F_{n-1} + F_{n+1})F_n = (2F_{n-1} + F_n)F_n. \end{aligned}$$

Now, to calculate the Fibonacci numbers for large n we can choose either the first equation (for odd numbers) or the second (for even numbers). We need to calculate two new Fibonacci numbers, but they are half the size of the original number. Repeating the process recursively, gives us a very fast algorithm to calculate large number: it is $O(\log n)$ Here is the python algorithm to do that.

```
def fib(n):
    if n==0:
        return 0
    if n==1:
        return 1
    if n%2==0:
        fn = fib(n/2)
        fn1 = fib(n/2-1)
        return (2*fn1+fn)*fn
    if n%2==1:
        fn = fib((n+1)/2)
        fn1 = fib((n+1)/2-1)
        return fn*fn+fn1*fn1
```

Code 41: Recursive python algorithm to calculate Fibonacci numbers.

End of lecture 4.