# Assignment 3 (python) (15 points)

All solutions should be stored in a directory called `assignment3` in your private repository.

## 3.1: File finder (5 points)

Make script which takes as argument a string and a directory. The script should search the directory recursively (meaning include all subdirectories, and subsubdirectories etc.) for files whose name contains the string, and print all those files to stdout.

Example usage:

```
$ python find.py .tex $HOME
/home/karlerik/work/INF3331/assignments/assignment_1.tex
/home/karlerik/work/INF3331/assignments/assignment_2.tex
/home/karlerik/work/INF3331/assignments/assignment_3.tex
/home/karlerik/work/INF3331/assignments/assignment_4.tex
```

Name of file: `find.py`

## 3.2: Unit tests for polynomials (3 points)

In the rest of this assignment, you are going to try test-driven development. The philosophy of test-driven development is, simply put, the following:

> before you start writing a piece of code, you should have an idea about what that code should do

The idea, then, is that *before* writing your code, you should write a `unit test` which confirms that your code actually has that property. For example, if you want to write an addition function `plus(a, b)`, you would expect that 2 and 2 becomes 4. This can be formalized as a unit test as follows:

```
def test_two_plus_two():
    assert plus(2, 2) == 4
```

A test should by convention have a name starting with `test_`, and raise an `AssertionError` (this is what the `assert` statement does) if the test fails. A disadvantage of this is that it is kind of boring. The advantages, however, is that it forces you to think clearly about what you want your code to do, and that if you do this, you end up with a test suite you can later use to make sure an update to your code doesn't accidentally break something which worked before.

In the next problem, you are going to make a Python implementation of normal, univariate polynomials $P(x) = a_n x^n + \ldots + a_0$. Your implementation should let you do things like the following:

```
p = Polynomial([1, 2, 3]) # the polynomial 3x**2 + 2x + 1
print(p(2))               # evaluates p at x=2
q = Polynomial([2, 6])    # the polynomial 6x + 2
print(p + q)              # should be 3x**2 + 8x + 3
```

Take a look at the stub `polynomials.py` to see which methods should be available. Before you start filling out the stub, however, write some unit tests. Implement the following tests:

- A test verifying that evaluating a polynomial of your choice at 3 different points gives what you'd expect
- A test verifying that adding two polynomials of your choice returns what it's supposed to
- A test verifying that subtracting two polynomials of your choice returns what it's supposed to
- A test verifying that each polynomial has a `.degree()` method which returns its degree

Name of file: `test_polynomials.py`

## 3.3: Polynomials (5 points)

Write a Python class `Polynomial` which lets you define a polynomial by giving a list of its coeffcients $a_i$. The `Polynomial` object should have methods `__call__`, `__add__`, `__sub__`, `__eq__`, `coefficients` `degree`. Sample usage of these methods and a stub of the `Polynomial` class is given in the example `polynomials.py`.

Name of file: `polynomials.py`

## 3.4: More polynomials (2 points)

Extend your code from 3.3 by adding methods `__repr__` and `__mul__`. The first of these should return a nice string representation of the polynomial, while the latter should let you multiply a polynomial by a normal `int`. Before doing so, add tests of these methods to `test_polynomials.py`. The tests should fail if your methods are implemented incorrectly, and pass if they are implemented correctly.

# Clarifications

## 3.1

- You may assume that all files and directories in the system has names without strange characters.
- You may assume that the filesystem is 'sane', in the sense that there are no broken or recursive symlinks.

## 3.2

- When writing the tests, you *don't have to* and in fact *shouldn't* create the *Polynomial* class. Instead, assume that it has already been defined in the file *polynomials.py*, and that your job is only to make sure it works properly. Your test functions will probably all throw exceptions all over the place until you complete assignment 3.3. That's ok and intended, because the code in it's current state *doesn't* work yet, so the tests *shouldn't* pass.
- Make sure your tests are reasonably rigorous! It is impossible to have tests which catch all possible errors, but that doesn't mean you should be lazy. When testing the `degree()` method, for example, you could just verify that the degree of $x^2 + 1$ is 2 and call it a day, but you could (and should) also make sure that the degree of $0x^3 + x^2 + 1$ is also 2 and not 3.
- We don't expect your tests to catch all possible errors, but you are expected to make an honest best effort to think about a reasonable way your program might fail and make a test which tests whether it fails that way.

## 3.3

- Methods like `__call__` are 'magic methods', meaning that they can be invoked with different kinds of syntax. So when you write `p(1)`, that really means `p.__call__(1)`. Similarly, `p+q` really means `p.__add__(q)`. This is useful for allowing users of your code to express themselves in a more natural way (`p+q` is a lot easier to read than `p.polyadd(q)`.)
- The `__rmul__` magic method is called when, in a term like `a*b`, `a.__mul__` doesn't know how to handle objects like `b`. In these cases, it tries to call `b.__rmul__` instead.

  For example, if you try to compute `3 * ''hei''`, Python first tries to call `3.__mul__(''hei'')`. When this fails, it then proceeds to calling `''hei''.__rmul__(3)`